1985

Top-down parsing syntax error recovery.

Hallowell, Paul E. Jr.

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

TOP-DOWN PARSING SYNTAX ERROR RECOVERY

by

Paul Evan Hallowell, Jr.

December 1985

Thesis Advisor: R. W. Floyd

Approved for public release; distribution is unlimited

Y CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

ORT SECURITY CLASSIFICATION lassified	1b. RESTRICTIVE MARKINGS
URITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
LASSIFICATION/DOWNGRADING SCHEDULE	
ORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)

ME OF PERFORMING ORGANIZATION l Postgraduate School	6b. OFFICE SYMBOL *(If applicable)* 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
RESS *(City, State, and ZIP Code)* terey, California 93943-5100		7b. ADDRESS *(City, State, and ZIP Code)* Monterey, California 93943-5100
ME OF FUNDING/SPONSORING ANIZATION	8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

RESS *(City, State, and ZIP Code)*

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

E *(Include Security Classification)*

-DOWN PARSING SYNTAX ERROR RECOVERY

SONAL AUTHOR(S)
owell, Paul E., Jr.

PE OF REPORT ter's Thesis	13b TIME COVERED FROM ________ TO ______	14. DATE OF REPORT *(Year, Month, Day)* 1985 December	15 PAGE COUNT 179

PLEMENTARY NOTATION

COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
LD	GROUP	SUB-GROUP	Top-down, syntactic error recovery, transition diagram parsing

TRACT *(Continue on reverse if necessary and identify by block number)*

iler writers continue to search for a reliable method of syntactic error
very. Spurious error reports and confusing diagnostics are common prob-
confronting the programmer. Innumerable error possibilities have made
very design a frustrating task.
thesis implements a method of syntactic error recovery using recursive
s on the error recovery routine. Parsing is accomplished by traversing
sition diagrams which are created from syntax charts. Key language
ols and dynamically generated recovery positions are used in restoring
parse. High-quality error diagnostics give a clear, accurate, and
rough description of each error, providing an excellent instructional
ware tool. Approach and implementation issues are discussed, and sample
ut listings are included.

TRIBUTION/AVAILABILITY OF ABSTRACT NCLASSIFIED/UNLIMITED □ SAME AS RPT. □ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED

ME OF RESPONSIBLE INDIVIDUAL el Davis	22b. TELEPHONE *(Include Area Code)* (408) 646-3091	22c. OFFICE SYMBOL Code 52Vv

RM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Top–Down Parsing Syntax Error Recovery

by

Paul Evan Hallowell, Jr.
Lieutenant Commander, United States Navy
B.S.M.E., United States Naval Academy, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

ABSTRACT

Compiler writers continue to search for a reliable method of ***syntactic error recovery.*** Spurious error reports and confusing diagnostics are common problems confronting the programmer. Innumerable error possibilities have made recovery design a frustrating task.

This thesis implements a method of syntactic error recovery using recursive calls on the error recovery routine. Parsing is accomplished by traversing transition diagrams which are created from syntax charts. Key language symbols and dynamically generated recovery positions are used in restoring the parse. High-quality error diagnostics give a clear, accurate, and thorough description of each error, providing an excellent instructional software tool. Approach and implementation issues are discussed, and sample output listings are included.

TABLE OF CONTENTS

I. INTRODUCTION

Syntax error recovery presents a most difficult challenge for the compiler writer. For a compiler to be a useful software tool, it must accurately recognize, analyze, and recover from syntax errors. The primary objective of syntactic error recovery is to permit the parsing mechanism to advance beyond the point of error detection in order to find and report subsequent errors to the programmer. Many strategies have been developed to recover from syntax errors, and while they may differ substantially in approach, they generally are concerned with the following goals:

(1) Detecting as many errors as possible

(2) Recovering from each error to permit parsing of the remaining text

(3) Generating thorough diagnostic information so that the user may fully understand the error

All syntactic recovery methods can detect the *presence* of at least one error, but none can guarantee a successful recovery from every error. Since it is impossible to know the intent of the programmer, it is imperative that compilers effectively *communicate* with the user by issuing accurate and informative error messages and minimizing spurious error reports. One of the major goals of this research is to improve the diagnostic aspect of syntax error recovery.

A. MOTIVATION

The parser detects a syntax error when the current input symbol prohibits the construction of a legal sentence in the language, i.e., the parser has entered a state from which it is unable to proceed. All detected errors fall into one of three categories: commission, omission, or substitution. An error of commission occurs when the parser encounters an extraneous lexical token which, if deleted, would result in a syntactically legal sentence. An error of omission means that inserting a lexical token into the input stream would yield a legal sentence. An error of substitution means that the parser has found an incorrect token; replacement is required to produce a valid sentence. Many strategies for recovery from syntax errors assume one of the situations above. Some techniques effect a *repair* of the error, via symbol insertions and deletions, while some search for a synchronization point from which the parser can regain control as if no error had occured. But which of the three kinds of errors is present? In some cases, determining the kind of error may not be difficult since the surrounding context provides information with which to analyze the error properly. However, consider the case where the *real error* occured much earlier in the source program and the detected error actually represents a *symptom* of the problem. In Pascal, for example, an extraneous "begin" in the middle of a program could remain undetected through several lines of code before a missing "end" is discovered. The same holds true for a deletion error where, for example, a missing "if x > y then" is actually the cause of an error which is detected later at "else". In situations such as these, the syntactic analyzer identifes the location of the error symptom, initiates a recovery, and outputs a message which is likely to be an erroneous or confusing description of the actual problem. More often than not, the parser loses synchronization, causing further problems with spurious errors, cascading error messages, and large portions of unparsed text.

Efforts to circumvent these problems take many forms. It is most difficult to design an error recovery scheme that blends recovery accuracy, security, and error message quality. The approach presented in this thesis seems promising in that regard. To establish a proper foundation for understanding the design, the following section reviews some of the previous efforts in syntactic error recovery.

B. BACKGROUND

Compiler error recovery methods are well documented in the literature. Since error recovery is a critical aspect of compiler design, many methods have been tried.

The most common form of syntax error recovery is a method referred to as the *panic mode*. This language independent technique is conceptually simple and easily applied to both top-down and bottom-up parsing algorithms. The scheme is based upon recovering only on a major terminating symbol, such as ";" or "end". Thus, if an error occurs near the beginning of a statement construct, for example, then text is discarded by the recovery routine until an end-of-statement token is recognized in the input stream. Although this method offers safety, its primary disadvantage is obvious: errors in the discarded text remain undetected. Despite the relatively primitive nature of the panic mode, the concept of synchronizing on key symbols is found in a number of different approaches.

Some of the earlier work in syntax error recovery concerns *minimum distance corrections.* This refers to the minimum number of symbol insertions, deletions, or replacements required to render an erroneous string valid. Aho and Peterson [Ref. 1] devised an algorithm that transformed strings in a time proportional to the cube of the length of the string by adding error productions to the language grammar. Lyon [Ref. 2] also investigated minimum distance error corrections using dynamic programming to choose from among possible corrections; however, these methods were mainly unfeasible. Levy [Ref. 3] simultaneously parsed potential correction paths from the point of error, one for each recovery possibility; however, the computations required often extend beyond a reasonable implementation limit.

Graham and Rhodes introduced an error recovery method called *phrase-level recovery* [Ref. 4]. This technique was initially configured for operator precedence parsing and later modified by Penello [Ref. 5] for use in LR analysis. Phrase-level recovery analyzes the error by examining its surrounding context, where the objective is to replace the phrase containing the error with a phrase that is syntactically valid. This is accomplished by a two-phase procedure consisting of a condensation (analysis) phase followed by a correction phase. The condensation phase involves bracketing the error context by means of a *backward* move, which attempts to perform further reductions on the stack, and a *forward* move, which endeavors to parse text beyond the location of the detected error to select the optimal repair. Although an accurate recovery is often possible with this approach, the primary disadvantage, as with all repair strategies, is that adequate repair becomes impossible if the parsing mechanism loses synchronization with the input stream.

Many error recovery schemes aim primarily at correcting single token errors, i.e., single errors of commission, omission, and substitution. However, one scheme which is oriented toward resolving a cluster of errors is discussed in Tai [Ref. 6]. This technique involves pattern matching forward of the error location, and is called a *k-correct lookahead corrector*. This means that *k* correct symbols must be found forward of the error to enable correction. Thus, each pattern represents a different string containing the error, where the closest pattern matching the input sequence is selected as the solution. Two major problems are inherent in this approach: the possibility of additional errors in the text forward of the detection point, and the fact that the choice of pattern used to effect the correction may depend on the symbol which follows a nonterminal whose expansion might involve a large number of tokens.

Ripley and Druseikis [Ref. 7] studied Pascal programming errors primarily to ascertain the validity of assumptions made by compiler writers in developing syntax error recovery techniques. One of the major results of this effort, based upon data obtained from several hundred student programs, was that most programming errors (almost 90%) are single token errors. Additionally, the observed error density was notably sparse, indicating that a recovery approach based upon repairing error clusters might not be the best choice. Thus, repairing errors local to the point of detection on the assumption that the damaged string represents a single error of commission, omission, or substitution appeared to be optimal in view of the study's results.

Fischer, Milton and Quiring [Ref. 8] developed an LL(1)-based *insertion only* algorithm, designed for implementation via a parser generator, where lexemes have associated *editing costs* which provide the basis for selecting the appropriate corrective action upon error detection. This notion of editing costs, or weighting values, emerged from the work of Graham and Rhodes [Ref. 4], in which the cost of symbol insertions, deletions or replacements corresponds to the number of changes required to the parsing stack to effect the repair. In the insertion-only technique, only the costs of *inserting* symbols are computed since deletion or replacement repair is not performed. Anderson and Backhouse [Ref. 9] improved upon this approach by using a *factorisation* lemma introduced by Backhouse [Ref. 10]. This lemma modified the recovery algorithm to compute the editing costs required to effect the *first* repair action instead of the complete repair. Thus, if the insertion of a three symbol string was required to restore the parse, the repair routine would be called three times before completely recovering from the error. This strategy reduces storage requirements and the size of the parsing tables at the expense of repeated calls to the repair routine.

The concept of editing the input string at the point of error detection was extended to include deletions and replacements in a *locally least-cost* error recovery approach [Ref. 11]. Implementation was accomplished via a parser generator which output a recursive descent analyzer based upon input BNF descriptions and editing-cost data for each terminal symbol in the grammar. This approach calls for string-edit operations based upon weighted values (cost) computed at point of error, and is applicable to LL(1) and LR(1) parsing algorithms, or any which possess the valid prefix property, i.e., report the presence of an error immediately after reading a symbol which does not permit continued parsing. One advantage of this method is that the costs may be modified either to create a certain level of recovery sophistication or to allow tailoring of recovery computations (editing costs) to take advantage of the most prevalent errors or error patterns. The primary disadvantage, however, is that since corrective action is strictly local to the point of detection, the wrong symbol may be inserted or deleted due to the absence of context information. Thus, an editing operation which is performed on an "error symptom" could be potentially diastrous.

Pai and Kieburtz [Ref. 12] also used local optimal syntax error repair but in conjunction with a *global context recovery,* thereby forming a two-level strategy. In this method, local repair is performed on a detected error, however, if this is insufficient, a global algorithm is invoked. Global context recovery discards tokens in the input stream until a *fiducial,* or trustworthy, symbol is encountered.

The stack is then adjusted to resume parsing beginning with this symbol. Barnard and Holt [Ref. 13] also discuss the use of synchronization symbols to perform *hierarchic* error repair. In this method, a separate synchronization stack holds potential recovery symbols for each nonterminal as it is being expanded during the parse. Should an error be detected, input is discarded until one of the synchronization symbols is found, at which point the parser is returned to a non-error state consistent with the chosen symbol.

Although many error recovery strategies are repair oriented, Richter has recently proposed a *noncorrecting* method of error recovery [Ref. 14]. In this technique, the symbol following the point of error detection is selected as the recovery point. The error is not corrected, but rather the remaining text is examined to determine whether a valid language suffix follows the error, in a process called "suffix analysis". The primary objective of this approach is to improve the accuracy and content of user error messages, and to prevent the generation of any spurious errors during the syntactic analysis. One shortcoming observed thus far, however, is that error detection of improperly nested contructs may be masked by the presence of an error that is internal to the scope of the construct.

In another non-repair strategy, Turba [Ref. 15] discusses an error recovery approach that parallels the exception handling mechanism in the Ada programming language. This technique has been implemented for LL(k) grammars in several programming languages, and is based upon user-defined recovery positions consisting primarily of the terminating symbols for each syntactic unit. Recovery sets are statically specified, and therefore do not necessarily correspond to the dynamic state of the parse at time of error. Thus, the potential exists to recover on the *correct* symbol in the *wrong* context. This method, while relatively similar to the panic mode, nevertheless takes advantage of more potential recovery points and avoids discarding large quantities of input while performing the recovery.

This thesis implements a top-down syntax error recovery method developed by R. W. Floyd. Although Floyd's approach is quite different from those discussed above, a few of the concepts mentioned, particularly the notion of fiducial symbols, have been embodied in the design. Syntactic analysis is performed by traversing transition diagrams, and the parsing and recovery mechanisms function recursively in response to detected errors. A complete discussion of the approach is presented in Chapter Two.

C. SCOPE OF THE THESIS

This thesis is an implementation of a Syntactic Analyzer that performs parsing and error recovery operations on Pascal programs. The Analyzer's processing capabilities include all syntax-related functions present in a full compiler implementation: lexical analysis, syntactic analysis (parsing), and syntactic and lexical stage error handling. Semantic analysis and code generation are not performed. The Analyzer accepts source program text, determines its syntactic validity, analyzes and recovers from detected errors, and outputs detailed diagnostics that identify and describe each error.

The design of the recovery scheme in the context of transition diagram parsing and the overall structure for the implementation were developed by R. W. Floyd. Software implementation of the Syntactic Analyzer, coding decisions, background research, and testing analysis are the accomplishments of the author.

D. IMPLEMENTATION STANDARD

The Syntactic Analyzer complies with the Pascal Language Standard approved by the International Standards Organization (ISO) in 1982 as "ISO 7185 Pascal Standard" [Ref. 16]. It must be noted that the Standard contains a provision for two versions of the language, Level 0 and Level 1 Pascal, where Level 1 incorporates the specification for conformant array parameters. The American National Standard (ANSI/IEEE 770X3.97-1983) is identical to Level 0 Pascal. The implementation in this thesis supports Level 1 Pascal.

Although Pascal was used to test the approach, the method described is not limited to Pascal. Parsing and error recovery algorithms are not dependent upon the implementation language.

E. THESIS ORGANIZATION

Chapter Two presents the design approach for both the parsing and error recovery mechanisms. Included are some examples of the actions performed during recovery that illustrate the recursive relationship between the normal execution and recovery modes of the syntax analysis. The basis for error message generation is also presented here.

Chapter Three discusses implementation considerations. The emphasis is on the components of the Syntactic Analyzer in terms of data structures, control structures, and program design decisions.

Chapter Four discusses testing of the syntactic analyzer and the strengths and weaknesses of the error recovery method when applied to Pascal programs. The appendices contain sample output listings, the diagram parsing specification, and the program listings with associated coding-level documentation.

II. APPROACH

The design approach for the Syntactic Analyzer is governed by two major objectives: to provide the user with accurate and thorough error diagnostic information and to detect as many source errors as possible to avoid repeated compilation. Error recovery design is based upon recursive calls to the error recovery routine, using intermittent returns to the parsing mode prior to recovering from the error. This method does not involve an insertion or repair strategy, but rather is consumption-based, discarding lexical tokens until a synchronizing symbol is encountered. Syntactic analysis is performed using the graphic design of language syntax charts to generate implementation data structures. Both parsing and error recovery operations are controlled by a stack, permitting recovery symbol generation to depend on each active context. The remainder of this chapter is devoted to the design and operation of both the parsing and error recovery mechanisms.

A. SYNTACTIC ANALYSIS

Syntactic analysis is accomplished by using stored language syntax diagrams to perform a top-down LL(1) parse of input text. Diagrams are traversed via an iterative controlling routine, using a parsing stack to hold nonterminal activation records during symbol expansion. Since the syntax diagrams are an integral part of the approach and form the basis for both syntactic analysis and error recovery, the concept of parsing from a diagram is discussed in detail below.

1. Diagram Structure and Composition

Syntax diagrams are nothing more than graphic depictions of the productions in the language grammar. They are composed of three entities: circular or elliptical figures, rectangular figures, and a series of connecting lines. The circular figures represent language terminal symbols, the rectangular figures denote nonterminal symbols, and the lines are *paths* which join the various syntactic units.

All information required to parse an input string is actually contained within the diagrams. The parsing and error recovery mechanism used here is guided by a *transition* diagram derived from the syntax diagram. The transition diagram may be thought of as a flow chart representation of its syntax diagram counterpart. Transition diagrams are formed from the syntax charts by specifying the paths of the charts as either true or false *exits* from each syntactic unit. Each nonterminal symbol is represented by a separate diagram. A transition diagram suitable for conducting parsing operations is created from a syntax diagram by ensuring that a deterministic path is provided at each branch point. The term *box* will be used to refer to the terminal and nonterminal symbols in a transition diagram.

2. Diagram Traversal

Parsing is accomplished by traversing the transition diagrams, following true or false exit paths from each box encountered. To explain how exit paths are labeled as true or false, we need to define some terms and illustrate their use.

Syntactic analysis is performed by an LL(1) parse of the input string. LL(1) means that the *next* symbol determines which production is followed where a choice between alternatives exists. A lexeme is *consistent* with a terminal box if it is identical to the lexeme associated to the box. A lexeme is consistent with a nonterminal box if it can occur as the first lexeme in a string derived from the nonterminal. A *true exit* from a box occurs when the box has consumed a string of the corresponding type. In particular, if a box is a terminal box, then a true exit occurs from this box after the single associated lexeme is consumed. A *false exit* from a box occurs when the first lexeme examined is not consistent with the box. In particular, if the box is a terminal box, then a false exit occurs if the current lexeme is not the lexeme associated to the box. The important point concerning a false exit is that no input is consumed. A third type of exit called the *error exit* is used to control error recovery. Error exit paths are not shown explicitly in the diagrams but their occurence is implied. An error exit occurs from a box if after consuming non-empty input, the box is unable to find valid input. A specific occurence of an error exit will be illustrated later in an example. The last term to define is *commitment*. When a box is entered, the current lexeme is found to be consistent and input is consumed. Once this occurs, we say that we are committed to a true exit from this box.

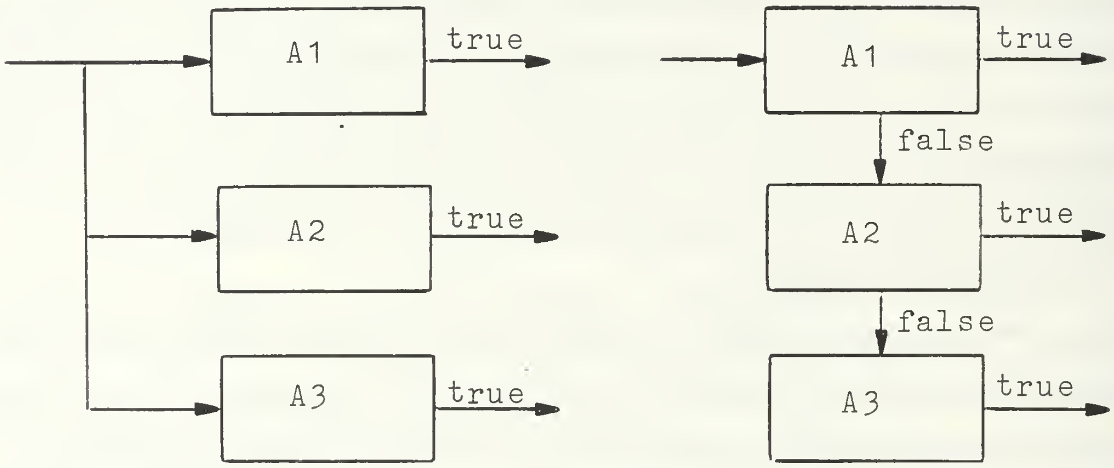


Figure 2.1 Syntax vs. Transition Diagram

Figure 2.1 illustrates the diagram convention. Notice the explicit representation of the true and false exit paths, where true paths leave boxes to the right and false paths emerge downward. Notice also how it is easier to visualize a false exit path from a transition diagram than from a syntax diagram. Remember, though, false exits do not indicate that the box was actually entered, but only that it was *examined* for entry. In Figure 2.1, if an instance of A1 is found, a true exit is taken and input is consumed; otherwise a false exit to A2 is taken and no input is consumed. If the first lexeme is consistent with A1, thereby eliminating A2 and A3 as alternatives, but an instance of A1 is not found, then an error exit is taken which is not shown explicitly.

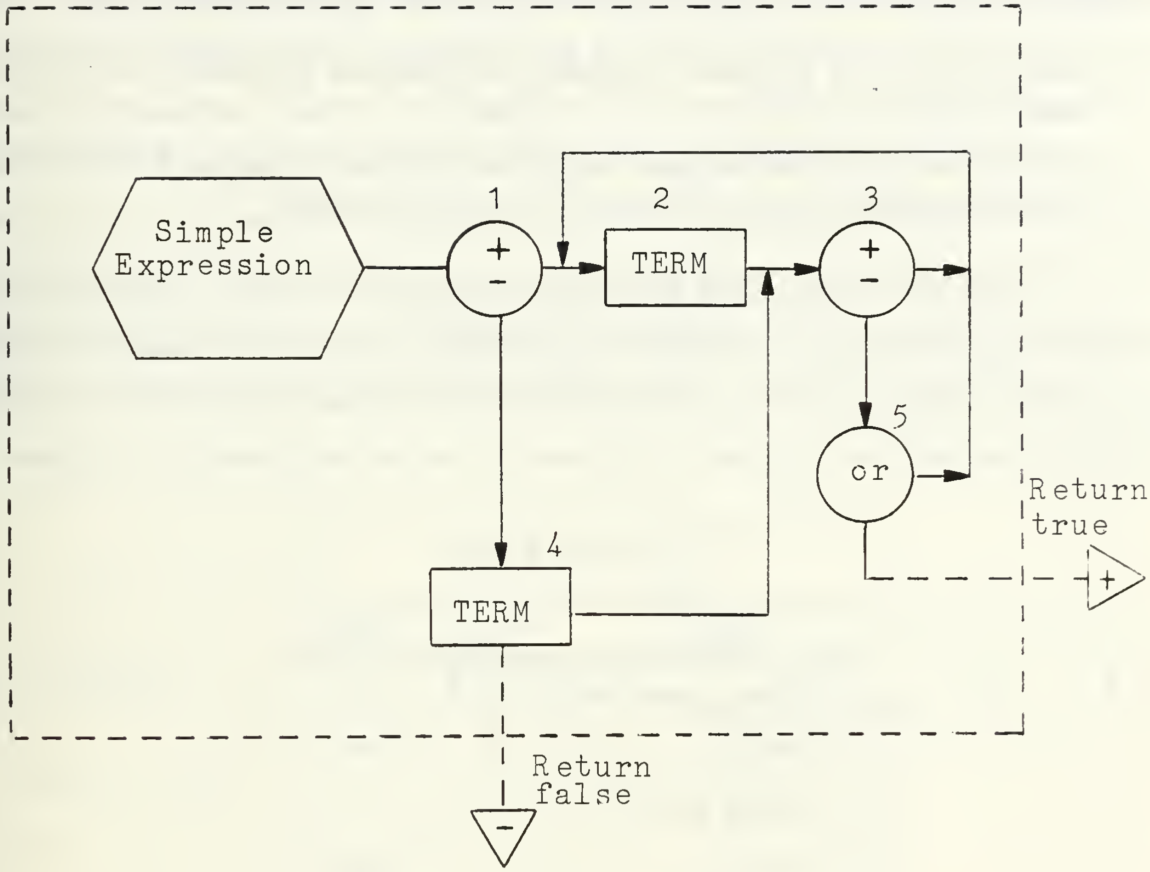


Figure 2.2 Transition Diagram for Simple Expression

Now let's see how a diagram is traversed. Figure 2.2 shows the transition diagram for Simple Expression. Notice the dotted box which encloses the diagram. This outer box is shown in order to relate a box of type A1 in Figure 2.1 to this illustration, i.e. we are effectively looking at the "inside" of a nonterminal box, where the nonterminal box *stands for* the corresponding diagram (to avoid infinite regress). Thus, parsing is accomplished by a series of *recursive* diagram calls. Notice in Figure 2.2 the larger arrowheads containing "+" and "-". These arrows correspond to the true and false exits shown above in Figure 2.1 for A1, where "+" is used for true and "-" is used for false. The reason for the initial downward extension on the false arrow from Box #5 will be discussed shortly. These exit paths, while true and false exits, have a special significance because they indicate points where diagram traversal will conclude. These will be referred to as *return true* and *return false*. The following definitions apply:

> *return true* -- the transition diagram has consumed a phrase of the specified type.
>
> *return false* -- the diagram, by inspection of the next lexeme, found without consuming input that no phrase of the type was present.

Now let's walk through the diagram in Figure 2.2 and see what can occur at each box. The key to understanding the diagram parse is to realize that each box must *uniquely* specify where to go for both true and false exits. A traversal table of true and false exit paths will assist the reader in following the diagram.

Traversal Table

Box	True	False
1 (adding operator)	2	4
2 (Term)	3	Error
3 (adding operator)	2	5
4 (Term)	3	Return false
5 (or)	2	Return true

Box #1 (adding operator) contains a true exit path to Box #2 (Term) and a false exit path to Box #4 (Term). Box #2 (Term) contains a true exit to Box #3 (adding operator) and an *error* exit if Box #2 finds no "Term" and takes a false exit. The only way an error exit can occur in Simple Expression is to consume input at at least one of the boxes, and then subsequently look for a Term when the current lexeme is not consistent with an instance of Term. It should be clear that terminal boxes have no error exits, although they may lead to error exits of enclosing nonterminal boxes. Boxes #4 and #5 contain the exits for Simple Expression. If an instance of Box #4 (Term) is not found, then traversal has completed in this diagram and control returns to the calling nonterminal box.

Box #5 is the only box in the diagram from which an instance of Simple Expression is reported as true to its calling nonterminal. This box is particularly interesting because a *false exit* from Box #5 ("or") results in a *return true exit* from the diagram. Earlier, we alluded to the arrow first extending downward and then to the right. This is because of the false exit from Box #5 followed by a return true exit from the diagram. Finally, the purpose of the two Term boxes deserves special mention. Note that an initial "adding operator" is optional since Term is the first box in the diagram from which a true exit *must* be taken in order to recognize an instance of Simple Expression. Now look at Figure 2.3.

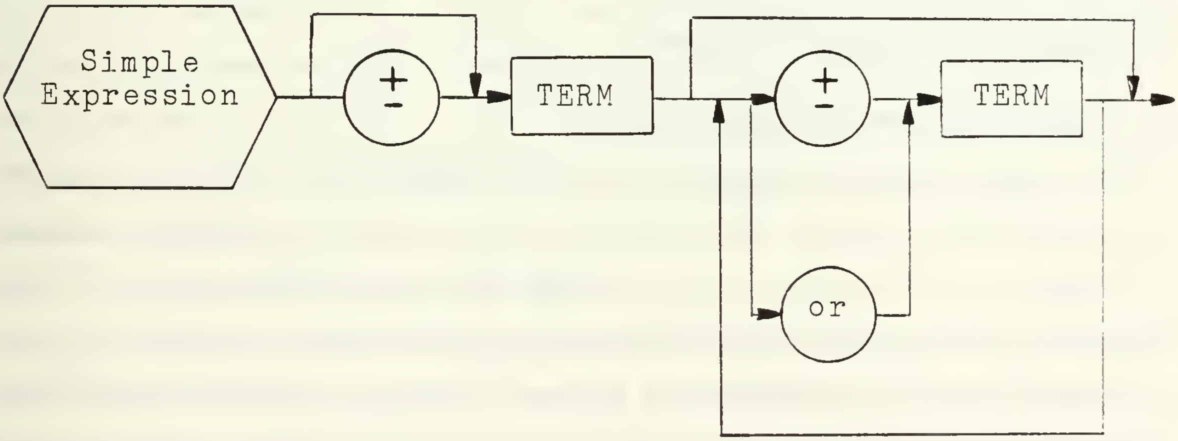


Figure 2.3 Syntax Diagram for Simple Expression

This is a syntax diagram for Simple Expression contained in Grogono [ref. 17]. Notice the optional path around "adding operator" and observe that if a false exit is taken from the leftmost Term box, there is no way to determine whether input has been consumed. Conversely, Box #4 in Figure 2.2 can only be reached if input has not been consumed (during the current traversal). Thus, Box #4 enables a *return false* on Term if input has not been consumed and Box #2 contains a false exit from Term if input has been consumed. This is typical of changes required to transform the syntax diagrams into deterministic transition diagrams. Diagram implementation changes are discussed in Chapter Three.

3. Normal Execution

To summarize diagram traversal and control, parsing is performed by a sequence of recursive calls on the transition diagrams which represent the nonterminal box expansions. A stack is used to hold nonterminal activations during diagram traversal, and transitions occur according to the exit criteria described above. When a nonterminal box is encountered, the header for the corresponding diagram is located and transitions through this new diagram continue until either a *return true* or *return false* condition is reached. Control then returns to the nonterminal box in the calling diagram from which the true or false path is followed based upon the exit condition. If an error exit is taken from a box, then the error recovery routine is invoked.

B. ERROR RECOVERY METHOD

As mentioned above, the error recovery strategy involves recursive calls to the error recovery routine. Error detection causes a recovery activation record to be placed on the parsing stack, invoking the error recovery routine. While recovery is active, input lexemes are discarded until a either a *resynchronization* or *restart* symbol is found (the set of recovery symbols is described below). If the symbol is a resynchronization symbol, the recovery activation record is popped, parsing mode is entered, and the recovery process is complete. If the symbol is a restart symbol, the recovery activation record is not popped, and the parsing mode is recursively entered, *suspending* the recovery process. Error recovery mode resumes when the recovery activation record becomes the top record on the parsing stack, continuing processing of the error which caused the initial entry into recovery mode from normal execution.

This method of error recovery offers several advantages. One is that more text will be parsed instead of discarded, permitting more errors to be detected. Another advantage is that cascading errors are avoided because potentially good text is not discarded while waiting for the "correct" symbol to appear (which may be several lexemes beyond a good restart point). A third advantage to this method is that the shared parsing/recovery stack, in conjunction with the recursiveness of the error recovery process, enables the syntactic analyzer to parse a large, heavily nested, error-laden language construct without risk of losing synchronization. The sections which follow describe the composition of the recovery symbol set, operation of the recovery mechanism, and generation of error diagnostic information.

1. Recovery Symbols

The contents of the recovery set is a key factor in determining the success of the error recovery. Two types of symbols comprise the recovery set: *resynchronization* symbols and *restart* symbols, which cause recursive entry into the parsing mode. All terminal boxes are potential recovery points in the transition diagrams.

a. Resynchronization Symbols

The set of resynchronization symbols is created from the stack of activation records upon entry into recovery mode following error detection. For each activation record on the stack, the corresponding diagram is examined for terminal symbols which are *reachable* by the paths from the box where the last true exit was taken. For example, in the erroneous segment:

```
var   next,last: integer,
```

where the error is "comma instead of semicolon detected after integer", the lexemes "," ":" and ";" would be resynchronization symbols, since they are the only terminal symbols reachable from the true exit of Type Denoter (see Figure 2.4); "var" is not a resynchronization symbol in this case.

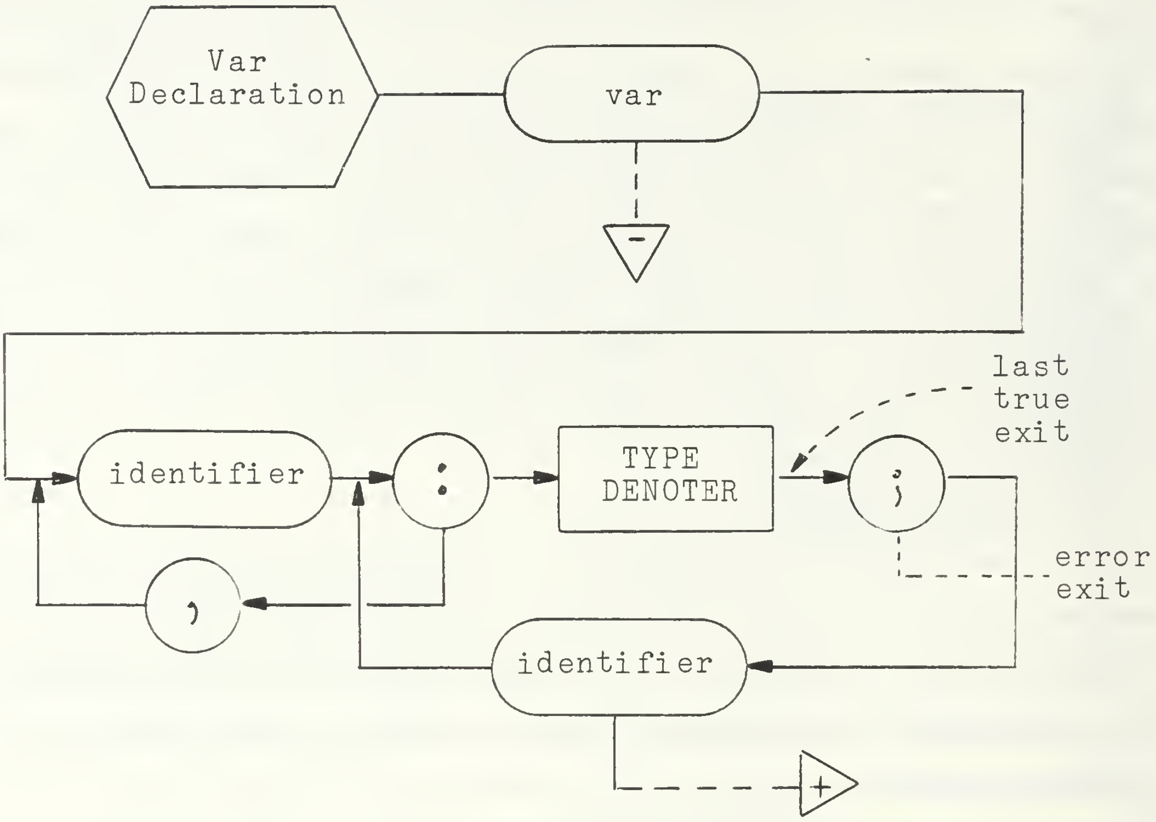


Figure 2.4 Transition Diagram for Var Declaration

Thus, searching the diagrams for recovery symbols is a matter of following true and false exit box paths to the end of the diagram. Since each diagram with an activation record on the stack is searched, the resynchronization component of the recovery set is the union of all resynchronization symbols which are reachable from the last true exit at any level of recursion. Should more than one recovery activation (and therefore, more than one recovery set) be present on the stack simultaneously, then the resynchronization set becomes a union of sets. Figure 2.5 depicts an erroneous Pascal code segment, the stack at time of error, and the symbols generated at each level.

```
program test;

begin
  x:= 1;
  if x > y > z then
    writeln(x)
  else
    writeln(y)
end.
```

Stack:

If Statement	---->	"then", "else"
Statement		
Compound Statement	---->	"end", ";"
Block		
Program	---->	"."

Figure 2.5 Stack During Error Recovery

The error in Figure 2.5 is an illegal "If Statement", detected by the parser at ">" following the Boolean expression "x > y". Notice that no symbols are generated for Block since only nonterminal boxes (Const Declaration, Var Declaration, and so forth) are contained in the diagram for Block, and also none are generated for Statement, which (in this case) only calls If Statement. Recovery occurs as soon as a lexeme in the input matches a symbol in the recovery set. Here, the recovery occurs at the Statement level on **then**. If the set were to contain any duplicates, such as two **else** symbols, then the symbol which is associated to the most recent stack activation would be selected for recovery.

b. Restart Symbols

Restart symbols cause a suspension in the recovery process and reentry into the parsing mode of syntactic analysis. These symbols are responsible for the recursiveness of the recovery process and for parsing rather than discarding text while performing a recovery. This set consists of symbols whose position in the transition diagrams is unambiguous -- specifically, any lexeme which occurs only once as a *first* symbol in a transition diagram. For example, **begin** occurs only once in the diagrams, as the first symbol in Compound Statement. However, **var** could signify either the beginning of Var Declaration or of the sequence "var x: integer..." in Formal Parameter List, and therefore is not a restart symbol. The recovery procedures associated with both the restart and resynchronization symbols are discussed later in this chapter.

2. The Recovery Mechanism

Entry into the recovery mode occurs either upon an error exit from the transition diagrams or when the top activation record on the parsing stack is a recovery activation from a previous error. In the latter case, resynchronization symbols have already been generated and the recovery simply "picks up where it left off". Otherwise, a new error has been detected, a recovery activation record is pushed onto the stack, and recovery set generation begins.

The operation of the recovery mechanism is illustrated by two erroneous Pascal programs. Consider the following code segment, which contains an error that demonstrates the two types of recovery mode operations:

```
program test;
begin
  if  x > y  than
    while x < z do
      x:= x + 1
  else
    begin...end;
  writeln
end.
```

Recovery mode is initially entered upon detection of the identifier "than", where the reserved word **then** was the required lexeme. The recovery set generated as a result of this error includes, among other symbols, the lexeme **else**, since it is a resynchronization symbol and it is reachable from the last true exit in the transition diagram for If Statement. Since "than" is an identifier (which is not a member of the recovery set), it is discarded by the recovery routine. The next lexeme delivered from the lexical analyzer is **while**, which is a member of the recovery set as a restart symbol. At this point, the recovery mode is suspended, an activation record for While Statement is pushed onto the stack, the transition diagram location pointer set to point at the **while** box, and normal execution (parsing) mode is re-entered. The stack upon resumption of the parse is shown below.

```
Top ------>

                  While Statement
                  If Statement(RECOVERY)
                  If Statement
                  Statement
                  Compound Statement
                  Block
                  Program
```

Notice that the recovery activation for If Statement is still on the stack, indicating that recovery for this nonterminal has not yet occured. After parsing While Statement, the old recovery record is now visible, causing a recursive call to the error recovery routine. Since the next lexeme is now **else**, and the previously generated recovery set for If Statement included **else**, recovery will occur immediately. The recovery record is then popped (since an error is not pending for this activation) and normal execution is reestablished.

Now let's examine a more complicated error sequence. The Pascal program shown above has been modified to create multiple errors, which will result in three pending recovery environments on the stack simultaneously:

```
 1  program test;
 2
 3  begin
 4    if x > y than
 5      while x < z doo
 6        begin
 7          x:= x + 1
 8          if x > 0 then
 9            z:= z - 1
10        end
11    else
12      begin...end
13  end.
```

The errors contained in the program above are as follows:

"than" instead of "then" in line 4
"doo" instead of "do" in line 5
a missing ";" in line 7

When the recovery routine encounters **if** in line 8, the stack is in the following configuration:

```
Top ----->

                    If Statement
                    Compound Statement(RECOVERY)
                    Compound Statement
                    While Statement(RECOVERY)
                    While Statement
                    If Statement(RECOVERY)
                    If Statement
                    Statement
                    Compound Statement
                    Block
                    Program
```

Syntactic analysis of this program results a sequence of transitions between the parsing and error recovery modes as listed below:

Recovery mode entered on "than" in line 4
Recovery mode suspended and parsing mode re-entered on **while** in line 5
Recovery mode entered on "doo" in line 5
Recovery mode suspended and parsing mode re-entered on **begin** in line 6
Recovery mode resumed on **if** in line 8
Parsing mode re-entered on **if** in line 8
Recovery mode resumed on **else** in line 11
Parsing mode re-entered and recovery mode complete on **else** in line 11

Upon recovering on the **else** in line 11, the recovery routine configures the stack to permit parsing to resume in the context of the **if** in line 4. This also pops the While Statement recovery activation, since the "while" construct is nested inside the "if" construct.

The two examples above typify the operation of the recovery mechanism. Chapter Four discusses several erroneous program segments to illustrate the effectiveness and accuracy of the error recovery method.

3. Error Messages

The primary objective of this approach was to implement a syntactic analyzer which could provide accurate and informative error diagnostics. By developing the syntactic analyzer using stored transition diagrams, the data required to generate high-quality error messages are readily available and obtainable from the boxes themselves. Because error messages are based solely upon information contained in the boxes, replacing or modifying transition diagrams has little or no effect upon the error handling routines. The following sections elaborate on the various components and procedures involved in the error computation and generation process. Implementation issues concerning error messages and error handler functions are addressed in Chapter Three.

a. History List

The *history* list is a collection of box names that represents the history of the parse within the current diagram at the time of error detection. This list corresponds to those box names (terminal or nonterminal) from which true exits were taken prior to entering the recovery mode. Thus, the following segment

```
begin
  x:= 1;
  if x > y then
    write(x);
  else...
```

would generate the following history list upon detecting the error "statement cannot start with "else":

```
begin <statement> ; <statement> ;
```

This information is available by accessing the top activation record on the stack (the current diagram being parsed). Each time a true exit occurs, the history list increases by one. Thus, the user is provided a narrative summary that is particularly useful in locating non-trivial errors or in finding errors that were actually made earlier in the code, such as in a large, heavily-nested compound statement.

b. Legal List

While the history list provides the user with a summary of correctly parsed constructs prior to error detection, the *legal* list is concerned with "what could have been". This list contains only terminal box names and consists of the Select set, or all of the permissible terminal boxes in the syntax which could immediately follow the box which represents the parser's last true exit prior to the error. Thus, in the Type Declaration segment

```
type length = ..60;
```

the following items below could immediately follow "=":

"identifier", "adding operator", "unsigned integer",
"unsigned real", "character string", "(", "^",
"packed", "array", "record", "set", "file" .

If a procedure block contained a "declaration out of order" error, such as

```
var i: integer;
type length = 40..60;
```

(where "type" must come before "var"), then the error would be detected at "type" and the legal list would consist of "procedure", "function", and "begin". The legal list is set empty whenever a true exit is taken and augmented by every terminal for which a false exit is taken.

c. Composite Message

The third component of user diagnostic information is the name of the diagram in which the error was detected, which is simply the name of the diagram for the activated recovery. So, combining the information components, the erroneous segment

```
procedure compute(x,y: integer): integer;
```

would yield the following error message:

```
Bad "proc/func declaration"
Recognized: procedure identifier <formal parameter list>
Legal would have been: ";"
```

In addition to the narrative diagnostic aid, a pointer to the source text marks the error location, and text discarded during the recovery process is underlined so that the user will readily see which portions of the program were affected. Additional discussion concerning these features and other error implementation issues are presented in the next chapter.

III. IMPLEMENTATION

The purpose of this chapter is to describe the primary modules of the Syntactic Analyzer in terms of major implementation decisions, data structure employment, and the function of key subroutines. Discussion is divided into four sections: lexical analysis, syntactic analysis, error recovery, and error message processing. Although this chapter is concerned with certain implementation details, specific coding-level and algorithmic comments are included with the program listings in Appendix C.

A. LEXICAL ANALYSIS

The first phase of compilation is lexical analysis, which provides the interface between the input and syntactic analysis phases, and concerns combining characters into single language units. The Syntactic Analyzer is configured for one-pass analysis; however, since co-routines are used to implement lexical and syntactic functions, lexical processing is discussed as a distinct phase. The input to the lexical anlayzer is a source program which is scanned as one continuous character stream, and the output is a sequence of lexical units called *lexemes*. This section defines the Pascal language symbols and constructs which comprise the lexeme set, and discusses the manner in which the input source text is processed in order to produce the lexemes.

1. Language Symbols

This implementation recognizes all word symbols, special symbols, and characters as defined by the Pascal Standard. The following describes the various units of the language which are forwarded to the syntactic analyzer as lexemes.

a. Word Symbols

All Pascal reserved words become lexemes. In addition to the thirty-four reserved words, the required procedures "write" and "writeln", as well as the directive "forward", are also included among the word symbols.

b. Special Symbols

All special symbols become lexemes. This category includes both single character symbols, such as '+' and '-', as well as multi-character symbols such as ':=' and '<>'. While all word symbols are given a unique lexical representation, not all special symbols are regarded as different lexemes, i.e., '<=' and '>' both generate the same lexeme since they are syntactically equivalent as a "relational operator".

c. Alternate Symbols

The Pascal Standard permits an alternate representation for selected symbols, i.e., '@' may be substituted for '^' to denote a pointer, and each alternate symbol is recognized by the Analyzer and processed as a lexeme.

d. Identifiers

Although some implementations may recognize an identifier at the syntactic level, it is formed here in the lexical stage. An identifier is a letter followed by zero or more letters or numbers in any combination.

e. Numbers

This category includes unsigned integers and unsigned reals. As with identifiers, real constants are not formed at the syntactic level. For example, 56.5 is not recognized as

<unsigned integer><period><unsigned integer>

but rather is recognized as

<unsigned real>

In order to permit lexical handling of errors which occur in specifying constants, an unsigned real number is recognized according to the following:

> any sequence built from digits, ".", "E", "E+", "E-", and not starting with "E" is treated as a (possibly illegal) number.

Thus, 1.23E-4+ will be recognized by the lexical analyzer as

<unsigned real><adding operator>

Processing of lexical stage errors will be addressed later in this chapter.

f. Character Strings and Illegal Characters

Any Pascal string constant becomes a lexeme. Any character scanned by the lexical analyzer (except those contained within comments and string constants) which is not a member of the Pascal Standard character set is recognized as an illegal character and will result in the generation of an illegal character lexeme. If successive illegal characters appear in the source text, then only one error lexeme will be produced, as in:

```
type word = pack$#%ed array[1..20] of char;
```

but the following will result in three illegal character lexemes:

```
type word = pac#k%ed# array[1..20] of char;
```

where the illegal characters in the preceding examples are: #, $, and % .

2. Lexical Analyzer Operation

The lexical analyzer, also known as the scanner, is divided into two major subroutines for processing source text. One routine is responsible for word recognition (anything beginning with a letter, which includes the reserved words and identifiers), and the second routine generates lexemes for all other symbols. The lexical analyzer communicates with the syntactic analyzer via a lexeme buffer. The lexical analyzer performs a character-by-character scan of input text, removing white space and line feeds until the packed group of character(s) forms a lexeme. Control then returns to the syntactic co-routine (parser). The following paragraphs briefly describe the structure and operation of the scanner's two lexical processing components.

a. Word Identification

A word buffer holds scanned input until the current input character is neither a letter nor a digit. Buffer contents are then compared against a stored array of reserved words. If a reserved word is found, the array index is returned as the lexeme; if it is not found, then an "identifier" lexeme is returned to the calling routine.

b. Symbol (non-word) Identification

The symbol identification section of the lexical analyzer is table-driven and simulates the operation of a finite state automaton. A two-dimensional array, indexed by current state and input symbol, is initialized with the required transitions for each input symbol/state combination. Transitions through the table continue until an accept state is reached, at which point the lexeme for that state is returned. The table generates lexemes for all symbols except identifiers and reserved words, and also filters any source text which is enclosed within comment symbols.

B. SYNTACTIC ANALYSIS

Syntactic analysis is accomplished by means of a top-down, deterministic traversal of transition diagrams derived from the syntax charts. Unlike recursive descent parsing, where separate routines are developed to process each nonterminal, this method is implemented with a stored transition diagram for each nonterminal and an iterative controlling routine. It is important to note that having the transition diagrams as data is essential to diagnostics and error recovery. As in predictive parsing, activation records are explicitly stacked; however, the records used here contain pointers into the transition diagrams. The following sections describe the structure and implementation of the diagrams and parsing mechanism.

1. Syntactic Analyzer Structure

The syntactic analyzer consists of two components: the transition diagrams and a parsing stack. The diagrams are represented by a set of records and the stack is implemented as a linked list.

As discussed in Chapter Two, diagrams contain boxes which represent language terminals and nonterminals. Each box corresponds to one record in the set and includes fields which specify box type, box name, lexeme code, true exit pointer, false exit pointer, and for nonterminal boxes, a pointer to the corresponding diagram. The parsing stack is implemented as a linked list of records, where each element of the list is an activation record for one nonterminal being parsed. Two kinds of records may be stack elements: one for normal execution and one for recovery operations. The following describes the information contained in each type:

Normal Execution:

a. return address -- the location of the parse (position within the transition diagram) when the activation record is created

b. diagram head -- a pointer to the header box of the active diagram

c. location pointer -- current box postion in the diagram set

d. last true exit -- the last box within the active diagram which was successfully recognized

e. history pointer -- a pointer to a linked list of all true exits taken in the diagram while the activation record is on the stack

Recovery:

a. diagram head -- used to identify the affected diagram for the error message

b. last true exit -- provides a starting point for recovery set generation

c. recovery set pointer -- a pointer to the set of recovery symbols

d. parent record pointer -- used to point at the level of stack that represents the diagram to which a recovery symbol belongs

2. Diagram Modifications

This section describes the changes required to the syntax diagrams to create transition diagrams that permit accurate error position identification and deterministic parsing. As we alluded to in Chapter Two, it is insufficient merely to extract published syntax drawings, create a box for each symbol, and create pointers for each line. A complete set of transition diagrams for Pascal is contained in Appendix B, and those boxes which pertain to the changes discussed here are clearly marked. Diagram modifications may be placed in the four categories described below.

a. Alternate Path Modifications

Changes in this category involve those diagrams which contain a box that can be reached in *two* ways, one of which consumes input while the other does not. Figure 3.1 depicts the difference between a syntax and transition diagram in representing alternatives.

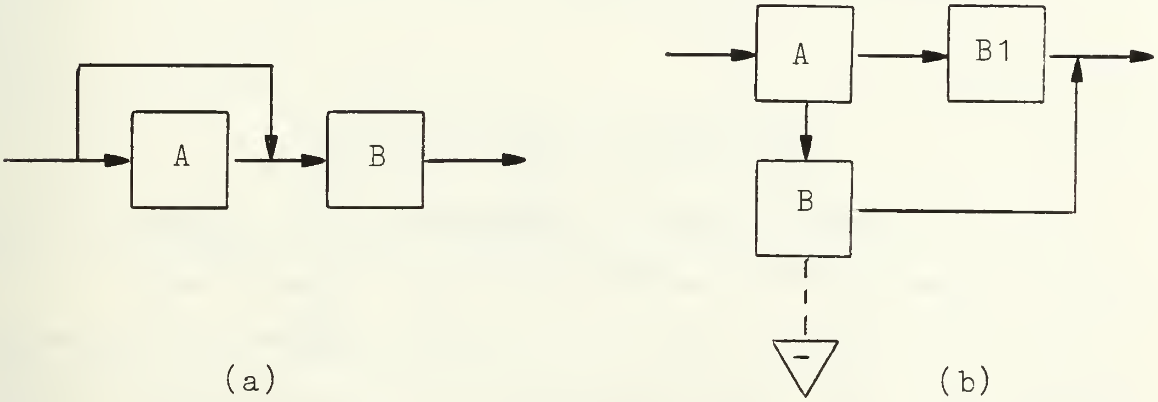
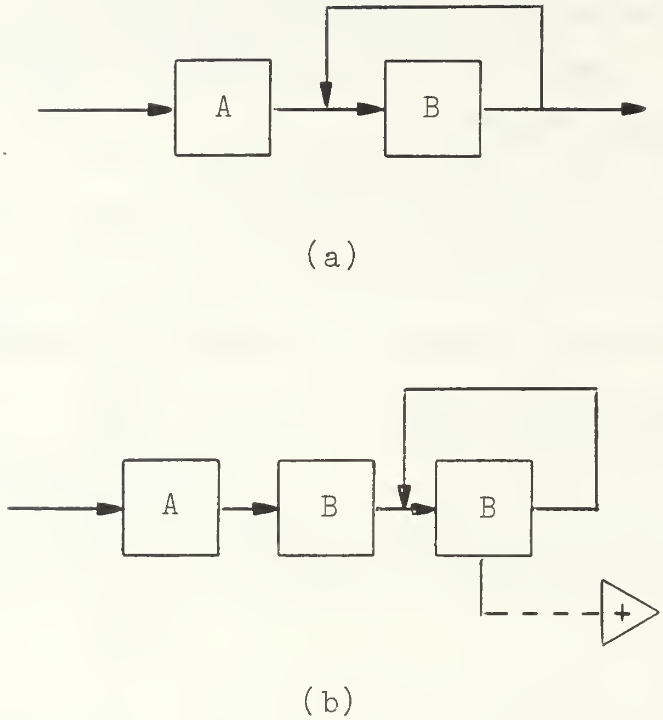


Figure 3.1 Alternate Path Modifications

Notice that at box B in the syntax diagram (3.1a), it is not possible to determine whether input has been consumed. Since parsing requires each box to have unique true and false pointers, a modification is required. By adding a box B1 in forming the transition diagram (3.1b), an error exit is taken from B1 if input was consumed, and a return false exit is taken from B if input was not consumed.

b. Looping Modifications

Changes in this category apply to those diagrams which permit multiple occurences, such as the Var and Type declaration parts in Pascal. This modification concerns those boxes which require at least one true exit, followed by zero or more true exits, prior to returning from the diagram. Figure 3.2 illustrates the modification required.



(a)

(b)

Figure 3.2 Looping Modifications

The syntax diagram (3.2a) provides no indication that at least one true exit was taken at box B. Conversely, the transition diagram (3.2b) shows that the first box B is required and that additional "loops" are optional. Thus, by adding another box, an error exit is taken if B is not found and a *return true* exit is taken if one or more occurences of box B are found.

c. Syntactic Modifications

The Analyzer, unlike a working compiler, does not retain the declared type of identifiers, and can't tell what symbols should follow an identifier. Since LL(1) requires that the next lexeme allow an unambiguous choice between alternatives, identifier boxes must be left-factored as shown below in Figure 3.3.

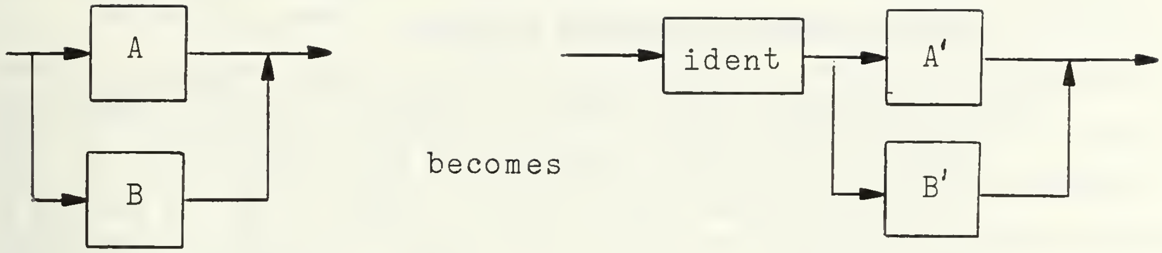


Figure 3.3 Factoring Modifications

d. Empty Statement Modifications

The existence of an *empty statement* in Pascal requires a special adjustment to the transition diagrams. If the empty statement is included as an alternate form of Statement, this violates the convention that a true exit implies input has been consumed. Normally an empty statement would be recognized by default if none of the Statement start symbols were found. But by specifying a return false from Statement and recognizing the presence of an empty statement in the calling diagram, the correct parsing structure is maintained and confusing error messages, which report successful recognition of an empty statement at a point where a statement start symbol is expected, are avoided.

3. Parsing Actions

Parsing begins when an activation record for the first diagram (Program) is pushed onto the stack. The location pointer is initialized to the first box in the diagram, and the lexical analyzer deposits the first lexeme into the lexeme buffer. Parsing from this point is simply a traversal through the transition diagrams, which advances based upon the following:

(1) If the location pointer points to a header box, then set the location pointer to the next box (first syntactic entity) in the diagram.

(2) If the location pointer points to a nonterminal box, then push an activation record onto the stack and set the location pointer to the header box of the appropriate diagram.

(3) If the location pointer points to a terminal box, then compare the contents of the lexeme buffer with the lexeme associated to the box. If they are identical, set the location pointer to the box specified by the true pointer and consume the lexeme; otherwise, set the location pointer to the box specified by the false pointer.

Parsing continues in this manner except when the location pointer is one of the following:

> Return true -- the current diagram has been successfully completed. Pop the stack and set the location pointer to the true pointer contained in the return address box.
>
> Return false -- no true exits were taken in the current diagram. Pop the stack and set the location pointer to the false pointer contained in the return address box.
>
> Exit error -- the buffer contains a lexeme which does not allow parsing to continue. Push a recovery record onto the stack and enter error recovery mode (discussed in the next section of this chapter).

Syntactic analysis concludes when the next lexeme is the end-of-file lexeme and the Program activation record is popped off the stack.

C. ERROR RECOVERY

Error recovery mode is entered for the purpose of resynchronizing the parse. As discussed in Chapter Two, there are two conditions which dictate a transition from normal execution: 1) recognition of a new error, and 2) the presence of a previous error recovery activation record at the top of the parsing stack, signifying completion of a restart phase. This section discusses the implementation of error recovery operations. Specific subroutine comments are included with the program listings in Appendix C.

1. Recovery Data Structures

Since the parsing stack is a dynamic structure, it follows that error recovery procedures should also function dynamically in restoring the state of the parse. The error recovery mode creates or accesses four dynamic list structures. One list is an error recovery tree, which is constructed and traversed in generating the set of recovery symbols. Two are linked lists which hold the resynchronization and restart symbols, and one is a list containing error records as nodes, where each node represents a separate error occurence and includes the various pointers which provide access to the message data. For clarification concerning the recovery sets described below, the term *recovery symbol list* refers to the set of resynchronization symbols which are dynamically generated following error detection. A *recovery set* consists of both resynchronization and restart symbols.

a. Recovery Tree

The recovery tree is a series of nodes which are created and traversed for the purpose of dynamically creating a set of potential recovery positions within the transition diagrams. Each node in the tree represents a diagram box which is reachable from the box that yielded the last true exit prior to error detection. A "depth first" search of the tree is performed to generate the recovery symbols.

b. Recovery Symbol List

The recovery symbols collected during the tree traversal are contained in the recovery symbol list which "extends" from the recovery record on the stack. The following information is stored in each node:

(1) symbol name

(2) lexeme code

(3) a pointer to the location of the symbol's box in the transition diagrams.

(4) a pointer to the activation record on the stack that represents the transition diagram which contains the box for this symbol.

When the buffer lexeme matches one of the lexemes in the list, parsing resumes at the box which is pointed to by the true exit pointer of the chosen symbol's box (#3 above). Since more than one recovery activation may be present on the stack simultaneously, a union of existing sets is formed by joining the list pointers, with the most recent list first. Figure 3.4 illustrates the parsing stack and a recovery symbol list which represents a union of symbols from pending recovery activations.

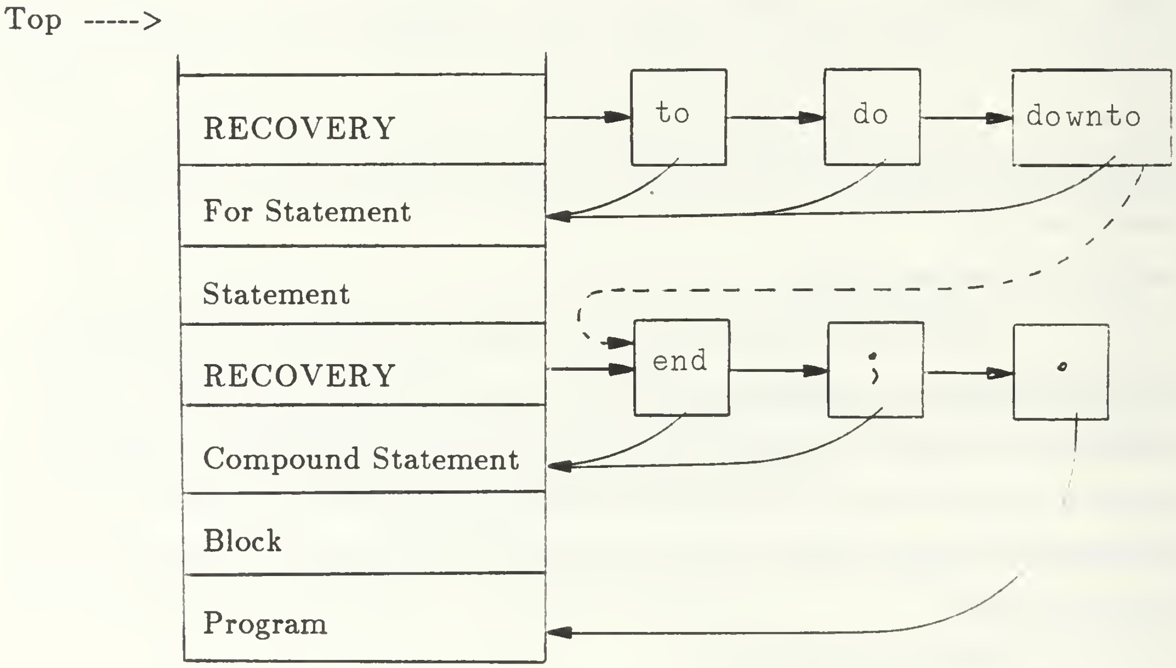


Figure 3.4 Recovery Symbol List

c. Restart Symbol List

This list is created during initialization of the transition diagrams. If a box has been designated as a restart lexeme, then a node containing this symbol is added to the list, along with the address of the diagram whose activation record belongs on the stack if the symbol is selected as a recovery point.

d. Error Record List

Once a recovery activation record has been pushed onto the stack, and prior to beginning the recovery process, a record of error information is created. This record contains the history list pointer, legal list pointer, source position pointer, and affected diagram name. This record then becomes a node in a linked list which contains all of the data for each error on the current source line.

2. Recovery Mode Operation

Three primary actions are required of the recovery module: generate the recovery set, search for a recovery symbol, and restore a normal parsing environment. If the recovery mode has been *resumed,* then only the latter two apply, since the previously generated set still remains as part of the old recovery activation record. The following briefly describes the implementation of these operations.

a. Generating the Recovery Set

Recovery set generation is implemented by means of a recursive controlling routine which builds and traverses the recovery tree in preorder (root-left-right). The recursion halts when either all diagram boxes (reachable from the last true exit) have been examined. This process is performed for each level of stack, i.e., the routine "walks down" the parsing stack, adding any symbol to the recovery list which has not yet been generated for the current activation.

b. Searching For a Recovery Symbol

Following recovery set generation, input is consumed until a recovery symbol is the next lexeme. Duplicate symbols may be present in the recovery set *only* if the set represents the union of two or more recovery lists (where the most recent, or nested, symbol would be selected). An error display handling routine is called to save the source positions of the "garbled" text (i.e., input which has been discarded during search) for later use in underlining the affected segments.

c. Restoring the Parse

If the recovery symbol is a restart symbol, then a new activation record is pushed onto the stack and parsing resumes within *that* diagram at the box pointed to by the true exit pointer of the restart symbol's box. Otherwise, activation records are popped off the stack (if required) until the correct record for the selected symbol is on top.

3. Lexical Errors

While the primary purpose of the Analyzer is to process syntactic errors, a brief mention is made here concerning lexical errors. Many lexical errors are corrected in the lexical analysis stage. If the scanner generates an invalid real constant error, for example, a lexeme adjustment routine is called to record the error (for later display with any syntactic messages) and modify the lexeme so that a valid real constant is returned. If an illegal character is detected, however, the error lexeme is passed onto the parser to permit the initiation of appropriate recovery action.

D. ERROR MESSAGE PROCESSING

As discussed in Chapter Two, the information needed to generate error messages is easily obtained by collecting the data during diagram traversal. The history list is updated each time a box true exit or a diagram return true exit is taken, adding a new entry for the lexeme or nonterminal box, respectively. The legal list is updated each time the buffer lexeme fails to match the lexeme associated to the box, i.e., upon every false exit from a lexeme box. Thus, the major portion of the message production process concerns those operations which are required for display formatting. As with the recovery routines, message display processing is performed almost exclusively using linked structures.

1. Error List Composition

There are three components or sources of error information: lexical stage errors, syntactic errors, and discarded text. Each error component is implemented as a linked list. In the syntactic error list, the nodes represent error records, one record for each syntactic error on the line, and contain the various error pointers such as the history list pointer. The lexical list contains the error position and a buffer with the text of the message. The discarded text list is a sequence of nodes, where each node contains start and stop source positions that bracket the corresponding text positions which require underlining.

2. Error Collection

The error handler is called by the end-of-line routine to output any messages for the line just completed. The lexical and syntactic error lists are merged to create an error sequence list for the line. Once the sequencing list has been created, it acts as a master controller, simultaneously traversing the syntactic and lexical lists and calling the output routine with the appropriate error record for display.

3. Line Formatting

All source text which is discarded by the error recovery process is underlined to provide the user with a clear indication of the Analyzer's recovery actions. Using the position information provided via the discarded text pointer, underlining is performed by creating a line buffer (array of characters) and assigning an underline character to each buffer position which coincides with a start-stop range in the discarded text list. Vertical dotted line formatting is also performed using the position information contained in the error sequence list. After each message has been output, the sequence list pointer is advanced one node, indicating that vertical line display also begins with the next position, thus creating the proper overlap required when multiple messages are displayed for a single line of text. Appendix A contains sample output listings which include examples of the various display effects when multiple error diagnostics are generated for a single line.

IV. TESTING AND DISCUSSION

The purpose of this chapter is to demonstrate the capabilities of the Analyzer through testing examples and to discuss recovery actions on representative errors. Since determining the effectiveness of an error recovery scheme is mainly subjective, we feel it best for the reader to draw his own conclusions.

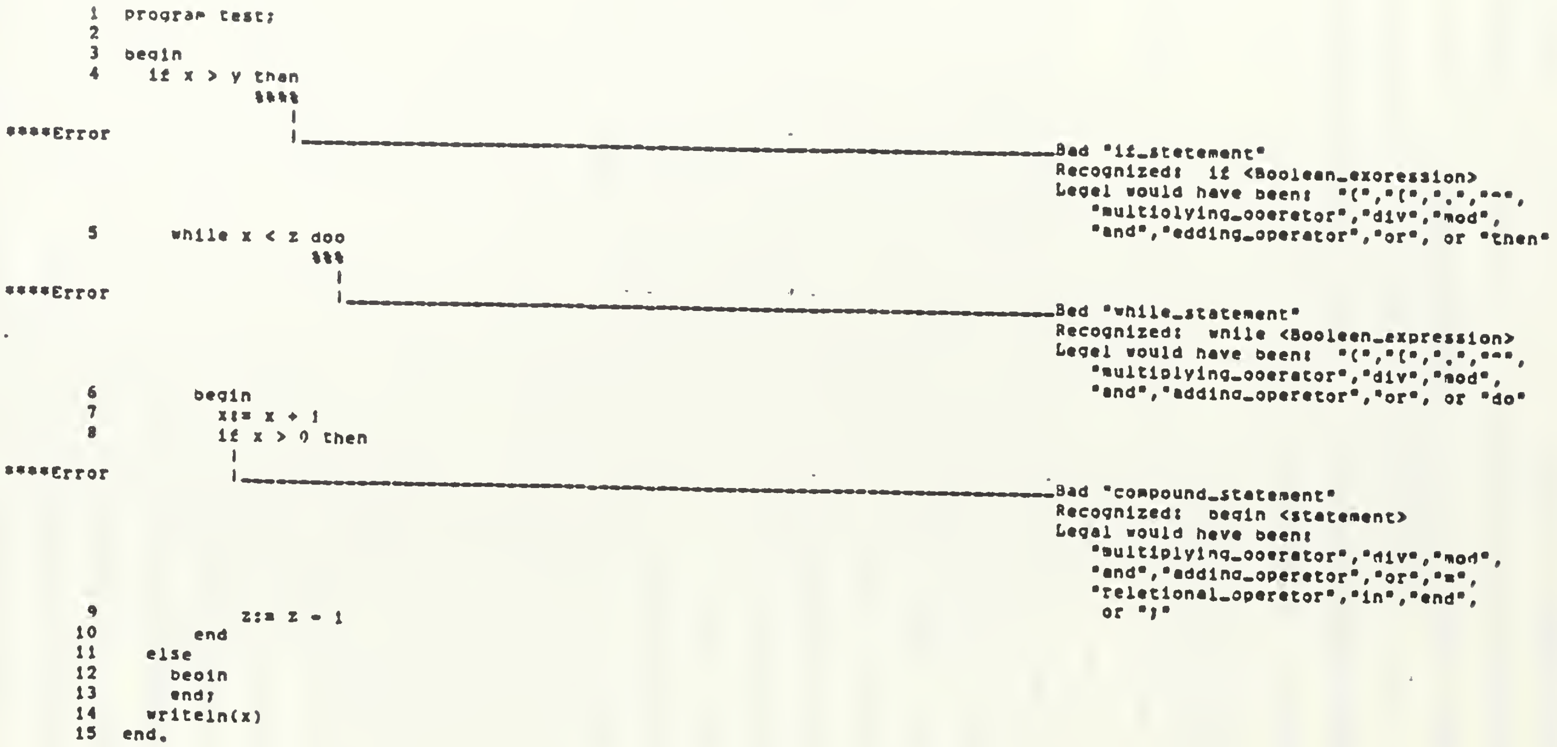
A. TESTING

The Syntactic Analyzer was tested using several Pascal programs. Many of these include representative erroneous text segments from the Ripley data base [Ref. 7], referred to in chapter one, while others were written by first quarter graduate students in an introductory programming course. Hand-constructed programs were designed to test Analyzer performance on code segments which contained numerous structural errors, and some Fortran programs were also run to further stress the recovery mechanism.

It is difficult to statistically measure error recovery effectiveness. Many researchers in the literature have used the Ripley program segments to test their recovery schemes and to serve as a basis for empirical analysis. While the segments were also used here, we feel that a more realistic assessment of Analyzer performance would be obtained by combining them into larger programs which contained the errors within several Pascal procedures. The programs used here each contain approximately 30 representative errors. Rather than attempt to categorize the recovery diagnosis in terms of excellent, good, etc., programs were examined only with respect to the ratio of error messages generated vs. minimum lexeme corrections, where minimum lexeme corrections is defined as the minimum number of lexemes required to transform the incorrect programs into syntactically valid ones. The sampling contained approximately 165 single lexeme errors which resulted in producing only 175 error messages. Although 6% of the messages were spurious, the induced messages were plausible and informative. For example, the illegal ";" in "if <Boolean expression> ; then..." resulted in one message for the If Statement as well as one subsequent message at **then** for the illegal beginning of a Compound statemnt. With an ideal ratio of one-to-one, the results are certainly encouraging. The next section will examine some of the output listings from these and other sample runs, and additional test listings are included in Appendix A.

B. REPRESENTATIVE CASES

Figure 4.1 contains the example program discussed in chapter two involving simultaneous recovery activations. Parsing initially halts on the identifier "than". The contents of the history list at the time of error detection are shown after "Recognized", followed next by those lexemes which would have been syntactically legal. Notice that the legal list contains many possibilities, as the identifier "y" could be part of a variable, the beginning of a larger arithmetic expression, or the beginning of a function call. Since "than" is not a resynchronization symbol, the text is underlined to show the user that it was discarded during recovery. The next lexeme, **while**, suspends the recovery process and parsing resumes with the pending If Statement recovery record on the stack. The next error is correctly caught at "doo" and, once again, no recovery occurs for the current activation since **begin** causes yet another restart by suspending recovery mode. By the time **if** is recognized in line 8, three recovery records have accumulated on the stack. At the **end** in line 10, parsing of the If Statement is completed and recovery mode is reentered to attempt resolution of the Compound Statement activation. Recovery occurs immediately on **end**, followed by a recursive recovery call at **else**. Although the While Statement recovery record is the top record at this stage, **else** is a member of the recovery set generated for the If Statement error. So, the recovery resolves the outermost error, and normal execution continues for the remainder of the program. Notice how little input was processed in the recovery mode. Although this example is relatively simple, it should be clear that the Analyzer frequently suspends and resumes the recovery process. With both the restart symbols



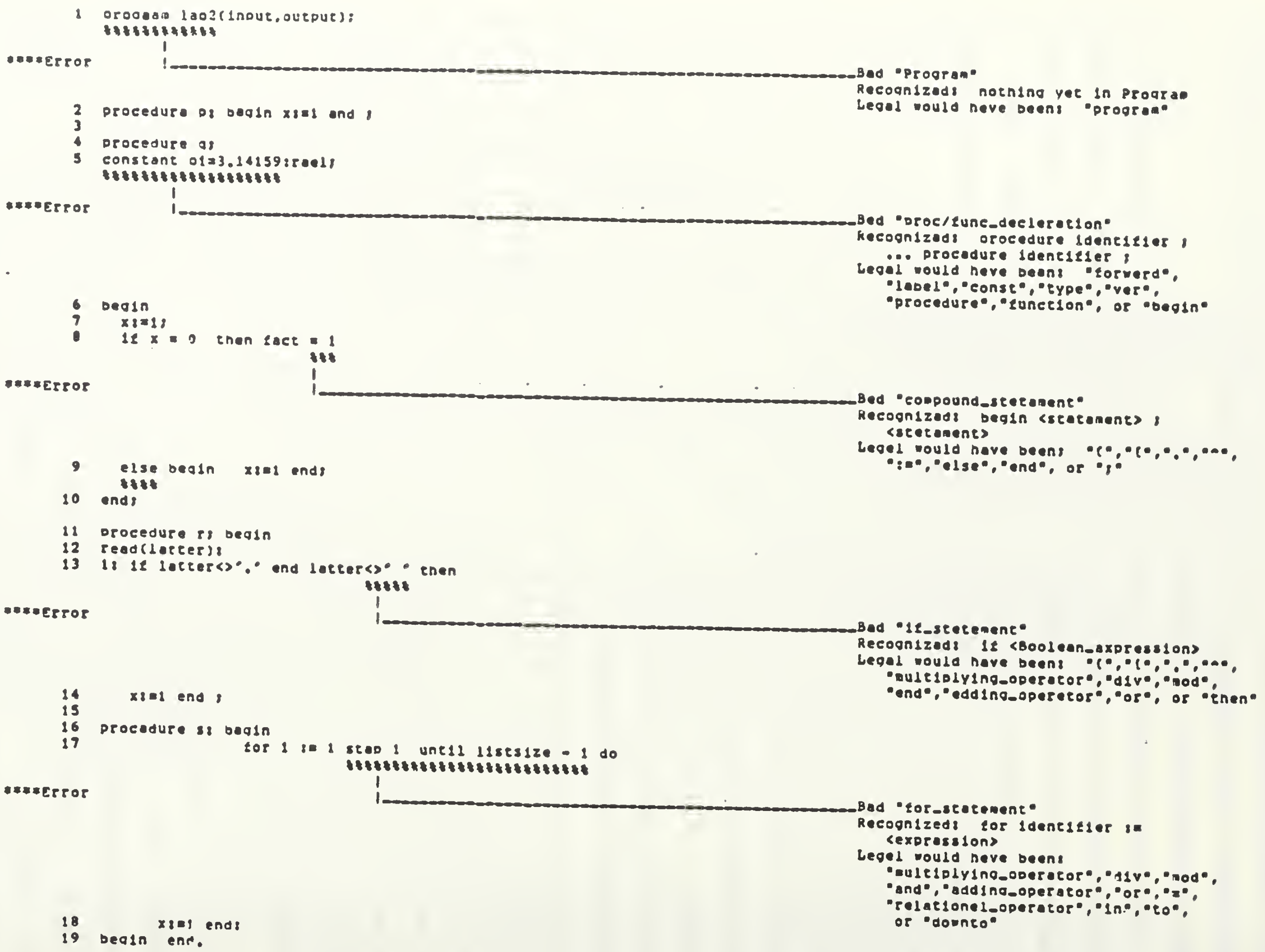
```
          1  program test;
          2
          3  begin
          4    if x > y then
                         $$$$
                            |
****Error                   |--------------------------------------Bad "if_statement"
                                                                    Recognized:  if <Booleen_expression>
                                                                    Legal would have been:  "(","[",".","--",
                                                                       "multiplying_operator","div","mod",
                                                                       "and","adding_operator","or", or "then"

          5      while x < z doo
                               $$$
                                 |
****Error                        |---------------------------------Bad "while_statement"
                                                                    Recognized:  while <Booleen_expression>
                                                                    Legal would have been:  "(","[",".","--",
                                                                       "multiplying_operator","div","mod",
                                                                       "and","adding_operator","or", or "do"

          6        begin
          7          x:= x + 1
          8          if x > 0 then
                     |
****Error            |---------------------------------------------Bad "compound_statement"
                                                                    Recognized:  begin <statement>
                                                                    Legal would have been:
                                                                       "multiplying_operator","div","mod",
                                                                       "and","adding_operator","or","=",
                                                                       "relational_operator","in","end",
                                                                        or ";"

          9            z:= z - 1
         10        end
         11    else
         12      begin
         13      end;
         14    writeln(x)
         15  end.
```

Figure 4.1 Sample Output Listing

and resynchronization symbols, less time is spent looking for a recovery point, more time is spent looking for additional errors, and fewer runs are required to obtain a syntactically correct program.

Figure 4.2 contains some sample program segments which demonstrate the Analyzer's recovery actions on common errors. Notice the error on line 8, where it appears that the user intended ":=" instead of "=". In this case, the error has caused the Analyzer to pop the activation for If Statement (as "fact" could be a legal procedure call), thereby eliminating **else** from becoming a resynchronization symbol. Nevertheless, the user is given an accurate description of what was recognized, since the last "<statement>" represents the If Statement and the discarded **else** is underlined. Detection of **begin** on line 9 initiates a return to the parsing mode, pushing a new Compound Statement *activation record* on top of the existing Compound Statement *recovery record.* When **end** is recognized, parsing of Compound Statement is complete and the "exposed" recovery activation record causes recursive entry into recovery mode, where the parse is immediately resynchronized on ";". This figure also shows examples of errors which were caused by misspelling of reserved words. Recovery after the identifier "progeam" occurs on "(", however, the recovery from "constant" (where **const** was expected) occurs on the ":" in line 5. This symbol was generated because of an existing Procedure/Function Declaration activation record on the stack, and it represents the symbol whose diagram box is part of a **function** heading sequence. Despite recovering on a symbol which did not belong to the Const Declaration diagram, the parse is back in step without any pending recovery activations.

The test segments contained in Figure 4.3 demonstrate recovery actions on an error of commission, omission, and substitution, as well as the integration of lexical errors in the error message output. Notice on line 5 that the illegal character messages from the lexical stage appear together with the syntactic error "bad write parameter list". The comment error at the end of the line, caused by the omission of a preceding "(", accurately informs the user that a "bad compound statement" was found. Each syntactically legal statement start symbol is provided in the message narrative, along with the two legal delimeters ";"



```
        1  program lab2(input,output);
           $$$$$$$$$$$$$
                 |
****Error        |---------------------------------------------Bad "Program"
                                                               Recognized:  nothing yet in Program
        2  procedure p; begin x:=1 and ;                       Legal would have been:  "program"
        3
        4  procedure q;
        5  constant pi=3.14159:real;
           $$$$$$$$$$$$$$$$$$$$$
                  |
****Error         |--------------------------------------------Bad "proc/func_decleration"
                                                               Recognized:  procedure identifier ;
                                                                  ... procedure identifier ;
                                                               Legal would have been:  "forward",
                                                                  "label","const","type","ver",
        6  begin                                                  "procedure","function", or "begin"
        7    x:=1;
        8    if x = 0  then fact = 1
                                 $$$
                                  |
****Error                         |----------------------------Bad "compound_statement"
                                                               Recognized:  begin <statement> ;
                                                                  <statement>
                                                               Legal would have been;  "(","[",".",",","^",
        9    else begin   x:=1 end;                               ":=","else","end", or ";"
             $$$$
       10  end;

       11  procedure r; begin
       12  read(letter);
       13  l: if letter<>'.' and letter<>' ' then
                                     $$$$$
                                       |
****Error                              |-----------------------Bad "if_statement"
                                                               Recognized:  if <Boolean_expression>
                                                               Legal would have been:  "(","[",".",",","^",
                                                                  "multiplying_operator","div","mod",
       14    x:=1 end ;                                           "end","edding_operator","or", or "then"
       15
       16  procedure s; begin
       17               for i := 1 step 1  until listsize - 1 do
                             $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
                                       |
****Error                              |-----------------------Bad "for_statement"
                                                               Recognized:  for identifier :=
                                                                  <expression>
                                                               Legal would have been:
                                                                  "multiplying_operator","div","mod",
                                                                  "and","adding_operator","or","=",
                                                                  "relationel_operator","in","to",
                                                                   or "downto"
       18       x:=1 end;
       19  begin  end.
```

Figure 4.2 Sample Output Listing

```
        1  program test(input,output);
        2  procedure p;
        3  begin
        4    if h<=1 then begin
        5        writeln(?error sort?,cntr,h,l,); goto 1; *abnrm*
                         $$$$$$$$$$$$$           |        $$$$$$$
                         |           |           |        |
****Error                |-----------|-----------|--------|-----------------------------------illegal cheracter(s)
                         |           |           |        |
****Error                |-----------|-----------|--------|-----------------------------------Bad "write_paremeter_list"
                                     |           |        |                                   Recognized:  (
                                     |           |        |                                   Legal would have baen:
                                     |           |        |                                      "edding_operetor","unsigned_integar",
                                     |           |        |                                      "unsigned_reel","cnarecter_string",
                                     |           |        |                                      "nil","identifier","[","(", or "not"
****Error                            |-----------|--------|-----------------------------------illegal charectar(s)
                                                 |        |
****Error                                        |--------|-----------------------------------Bad "write_parameter_list"
                                                          |                                   Recognized:  ( junk , ... , <expression>
                                                          |                                      ,
                                                          |                                   Legal would have been:
                                                          |                                      "edding_operetor","unsignad_integar",
                                                          |                                      "unsignad_real","cherecter_string",
                                                          |                                      "nil","identifier","(","(", or "not"
****Error                                                 |-----------------------------------Bed "compound_stetement"
                                                                                              Recognized:  bagin <statamant> ;
                                                                                                 <stetement> ;
                                                                                              Legal would heve bean:
                                                                                                 "unsigned_integer","identifier","goto",
                                                                                                 "write","writeln","begin","if","cesa",
                                                                                                 "repeet","while","for","with","end",
                                                                                                  or ";"

        6      end ; x:=1
        7  and ;
        8
        9  procedure q;
       10  begin
       11    writeln('  ',t,'   ',list[t]);
       12    end;
       13  end;
           $$$$
             |
****Error    |--------------------------------------------------------------------------------Bad "block"
                                                                                              Recognizen:  <proc/func_decleration>
                                                                                              Lagel would have baen:  "procedure",
                                                                                                 "function", or "bagin"

       14
       15  procedure r;
       16  type alfe = packed array [1-10] of cher;
                                           $$$
                                            |
****Error                                   |-------------------------------------------------Bad "ordinal_type"
                                                                                              Recognized:  <constant>
       17  begin x:=1 end ;                                 begin  end.                       Legel would heve been:  ".."
```

Figure 4.3 Sample Output Listing

and **end**. The second error for Write Parameter List in line 5 contains the term "junk". This corresponds to the previously discarded text and was inserted into the history list in order to accurately reflect the cumulative status of the parse for this construct. In line 13, the Analyzer detects an error of commission where an **end** with no matching **begin** is found. The **end** is discarded and the message indicates that a complete procedure block has been recognized where either the beginning of a Compound Statement or another Procedure/Function Declaration was expected. Finally, the error on line 16 shows a substitution error, where the user is informed of the *only* symbol which would have been syntactically legal following a preceding <constant> in Ordinal Type.

Not all recoveries were performed as easily as those discussed above. Figure 4.4 contains two examples which show errors that generated more than one message. The sequence in line 2 results in three recoveries within the Formal Parameter List activation. Parsing terminates at "," where ";" was expected, and recovery occurs on the same lexeme. The ensuing error at **var** is due to the previous recovery which restored the parse in the middle of an "identifier list", and the second erroneous "," also leads to recovery on the same lexeme. All four recoveries on this line are performed correctly in terms of resuming at the proper transition diagram box, but only three incorrect lexemes are present. Although an extra message was generated, no text was discarded and the messages provide a clear indication of exactly what was expected and what action was taken. In line 6, the error is correctly diagnosed, but recovery occurs on the ";" which represents the box that terminates a procedure or function heading. The identifier "boolean" is then regarded as either the lexeme **forward** or a Block nonterminal, where the parse resynchronizes at the ";" corresponding to the end of a Procedure/Function Declaration. Thus, the subsequent message states that a "Bad block" has been found, and the Analyzer returns to normal execution at **begin**. Nevertheless, as in the Formal Parameter List example above, the user is provided with a clear display of recovery actions.

```
       1  program p(input,output);
       2  function toosort(var x : order, var y : sorted, x: integer);
                                        |   |          |              |
****Error                               |---|----------|--------------|--------------------Bad "formal_peramater_list"
                                            |          |              |                    Recoonized:  ( ver identifiar :
                                            |          |              |                       identifier
                                            |          |              |                    Lagel would have baen:  ")", or ";"
                                            |          |              |
****Error                                   |----------|--------------|--------------------Bad "formal_paramater_list"
                                                       |              |                    Recognized:  ( ver identifier :
                                                       |              |                       identifier ,
                                                       |              |                    Legal would have baen:  "idantifier"
                                                       |              |
****Error                                              |--------------|--------------------Bad "formal_peramater_list"
                                                                      |                    Recognized:  ( var identifiar ...
                                                                      |                       identifier : identifier
                                                                      |                    Lagel would hava baan:  ")", or ";"
                                                                      |
****Error                                                             |--------------------Bed "proc/func_daclaration"
                                                                                           Recognized:  function identifiar
                                                                                              <formal_paramater_list>
                                                                                           Legal would hava baen:  ";"

       3  beoin
       4  end;  (toosort)
       5
       6  function getelemant(var x : integer) ; booleen ;
                                               | %%%%%%%
                                               |        |
****Error                                      |--------|----------------------------------Bed "proc/func_daclaration"
                                                        |                                  Recognized:  function identifier
                                                        |                                     <formal_parameter_list> ... function
                                                        |                                     identifier <formel_peramater_list>
                                                        |                                  Legel would hava baen:  ";"
                                                        |
****Error                                               |----------------------------------Bed "proc/func_declaration"
                                                                                           Recognized:  function identifier
                                                                                              <formal_oarameter_list> ...
                                                                                              identifier <formal_peremater_list> ;
                                                                                           Lagel would hava baen:  "forward",
                                                                                              "label","const","type","var",
                                                                                              "procedura","function", or "bagin"

       7  ver q:integer;
          %%%%%%%%%%%%%%
             |
****Error    |-----------------------------------------------------------------------------Bed "block"
                                                                                           Recognized:  <proc/func_daclaration>
                                                                                           Lagel would have been:  "procedure",
                                                                                              "function", or "bagin"

       8  beoin  x:=1  end;
                           %
       9
      10  procedure p;
      11  type order array[1..maxreletions,1..2] of integer;
                     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                          |
****Error                 |----------------------------------------------------------------Bad "type_declaration"
                                                                                           Recognized:  typa identifier
                                                                                           Legal would hava baen:  "="

      12  var o:integer;
      13  begin x:=1 end;
      14
      15  begin x:=1 end.
      16
```

Figure 4.4 Sample Output Listing

C. DISCUSSION

Based upon testing performed thus far, it appears that the use of restart symbols to control recursive calls to an error recovery routine is practical, reliable, and effective. Rather than pursue a recovery mode solution for each detected error, it seems advantageous to suspend the recovery process upon recognizing a trustworthy symbol, traverse the diagram which begins with this symbol, and then return to resume the recovery. Thus, in a program which contains several errors, parsing is actually accomplished incrementally, moving from one segment which begins with a restart symbol to another. Each time the recovery process is suspended, the parser is able to detect any errors which may be present in the new segment, ultimately analyzing most of the text and possibly detecting *all* of the errors. Although several pending recovery records may remain "unresolved" on the stack, the end result is that synchronization is maintained and propagating error side effects, which cause confusing and unnecessary messages, are eliminated.

One reason for the success of this method is that the restart symbols appear both frequently and conveniently separated in a typical program. In Pascal, all of the declaration start symbols, with the exception of **var**, are members of the restart set. Recall that **var** may appear in either a declaration part or a formal parameter list and, therefore, provides an ambiguous resumption point. So, there exists a kind of "protection" against losing step no matter how serious the error or combination of errors may be (assuming that the resynchronization set hasn't already provided a symbol upon which to resume). Similarly in the compound statement portion of a program, where almost all of the statement start symbols are members of the restart set, protection is provided against a prolonged search for a recovery point. Thus, the restart symbols are not only trustworthy from the standpoint of providing an unambiguous position within the transition diagrams, but they always seem to be in "just the right places". Combining these symbols with resynchronization recovery points from the active contexts, the end result is that more errors have been detected.

While the restart symbols are the key to the recovery scheme, the resynchronization symbols provide not only additional recovery points, but also an element of safety as well. Since only positions reachable from the last true exit in the active diagrams are chosen, some potentially good recovery points may be excluded. Line 11 in Figure 4.4 shows an invalid declaration where the error is correctly identified as "missing =". Although **array** would appear to be a good recovery point in this context, recovery does not occur until the delimiting semicolon is recognized, as shown by the underlined text. This is because the error occured in the Type Declaration context and an activation for Type Denoter has yet to be pushed onto the stack. Thus, symbols such as **packed**, **array**, etc. are not members of the recovery set since the resynchronization symbols are derived only from the stack configuration at time of error detection. During the initial phases of implementing this recovery method, some experimentation was performed in attempting to effect a recovery in fewer lexemes by building on the stack after pushing a recovery record. In other words, the nonterminals from the active diagram that are reachable from the last true exit would be expanded to provide additional recovery possibilities. But the larger size of the recovery set and the risk of recovering in the wrong activation ultimately resulted in inducing extra errors.

The most significant characteristic of this recovery scheme is the quality of the error messages and its value as an instructional software tool. If the primary goal of a compiler is to effectively communicate with the user, then this approach seems to have lived up to standards. Cascading error messages have been eliminated and each message provides only the facts about what "was" and "what could have been". The novice programmer is undoubtedly a primary beneficiary. Between the history list, the syntactically legal list, source position pointer, and the underlining of discarded text, the user is certainly provided with enough information to fully understand the error and the actions performed by the Analyzer during the recovery. In the erroneous Pascal sequence, "if...then begin...end ; else...", many compilers would issue a message similar to " ; can never come before **else**". While this accurately describes the problem, a diagnostic which explains that **else** cannot occur after the sequence "<statement> ;" in a compound statement is much clearer. It specifically states, in the context of the language syntax, that a statement (If Statement) has been recognized and that a new statement cannot begin with **else**. The combination of the three diagnostic aids (error message, source pointer, underlining) leaves little room for any misunderstanding of reported errors. If the complete diagnostic package is undesirable for a more advanced user, incorporation of a "help" selection feature could provide the means for tailoring the output to the requested level of assistance.

D. SUGGESTIONS FOR FUTURE EFFORTS

This thesis is a step toward determining the effectiveness and usefulness of this method of error recovery. Testing results appear to confirm its feasibility; however, further testing needs to be performed and should include experimentation with various recovery set combinations to ascertain an improved configuration. While efforts thus far have been directed at the syntactic level, a longer term objective should be to incorporate the Syntactic Analyzer into a full compiler implementation, where a first pass would generate syntactic error messages and a second pass would add the semantic errors. Thus, the error messages could be integrated in the output as was done here with the lexical and syntactic messages. Although this implementation was performed for Pascal, future efforts might explore the feasibility of this approach for other higher level languages. The syntax diagram traversal concept seems easy to extend, and many languages contain a number of symbols which could be designated as "fiducial" for recovery purposes. Certainly, programmers of all languages would benefit from reliable error recovery and informative diagnostics.

APPENDIX A: SAMPLE OUTPUT LISTINGS

```
      1
      2
      3   program sample1(input,output);
      4
      5   procedure x; begin
      6                     a := 0; null := "          ";
                                            $$$$$$$$$$$$
                                            |.         |
****Error                                   |----------|-----------------------------------illegal character(s)
                                            |          |
****Error                                   |----------|-----------------------------------Bad "statement"
                                                       |                                   Recognized:  identifier :=
                                                       |                                   Legal would have been:
                                                       |                                      "adding_operator","unsigned_integer",
                                                       |                                      "unsigned_real","character_string",
                                                       |                                      "nil","identifier","[","(", or "not"
****Error                                              |
      7                 end ;                          |-----------------------------------illegal character(s)
      8
      9   procedure p; begin
     10                  read(letter);
     11                  1: if letter<>'.' and letter<>' ' then
                                                      $$$$$
                                                       |
****Error                                              |-----------------------------------Bad "if_statement"
                                                                                           Recognized:  if <Boolean_expression>
                                                                                           Legal would have been:  "(","[",".","^",
                                                                                              "multiplying_operator","div","mod",
                                                                                              "and","adding_operator","or", or "then"
     12                     x:=1
     13                end ;
     14
     15   procedure p; var
     16                label, loc, index, rsize: integer;
                           || $$$  $$$$$  $$$$$
                           ||  |     |      |
****Error                  ||--|-----|------|----------------------------------------------Bad "var_declaration"
                            |  |     |      |                                              Recognized:  var
                            |  |     |      |                                              Legal would have been:  "identifier"
****Error                   |--|-----|------|----------------------------------------------Bad "label_declaration"
                               |     |      |                                              Recognized:  label
                               |     |      |                                              Legal would have been:  "unsigned_integer"
****Error                      |-----|------|----------------------------------------------Bad "label_declaration"
                                     |      |                                              Recognized:  label ,
                                     |      |                                              Legal would have been:  "unsigned_integer"
****Error                            |------|----------------------------------------------Bad "label_declaration"
                                            |                                              Recognized:  label , junk ,
                                            |                                              Legal would have been:  "unsigned_integer"
****Error                                   |----------------------------------------------Bad "label_declaration"
                                                                                           Recognized:  label , junk , junk ,
                                                                                           Legal would have been:  "unsigned_integer"
     17                begin
     18                  x:=1
     19                end ;
     20
     21   begin  end.
```

```
        1
        2
        3 program sample2(input,output);
        4
        5 procedure p;
        6              begin if list[index] < list[loc] ;
                                                        |
****Error                                               |--------------------------------Bad "if_statement"
                                                                                         Pecognized:  if <Boolean_expression>
                                                                                         Legal would have been:  "[",".","^",
                                                                                             "multiplying_operator","div","mod",
                                                                                             "and","adding_operator","or", or "then"
        7                      then x:=1
                               $$$$$$$$$
                                  |
****Error                         |------------------------------------------------------Bad "compound_statement"
                                                                                         Recognized:  begin ;
                                                                                         Legal would have been:
                                                                                             "unsigned_integer","identifier","goto",
                                                                                             "write","writeln","begin","if","case",
                                                                                             "repeat","while","for","with","end",
                                                                                              or ";"
        8              end ;
        9
       10 procedure p; begin
       11                pack(buf,1,a);
       12                list(t):= a;
                                $$$$
                                 |
****Error                        |-------------------------------------------------------Bad "compound_statement"
                                                                                         Recognized:  begin <statement> ;
                                                                                             <statement>
                                                                                         Legal would have been:  "end", or ";"
       13                x:=1
       14              end ;
       15
       16 procedure p;
       17              var key,record : array[1..limit] if akfa;
                               $$$$$$                    |     |
                                    |                    |     |
****Error                           |--------------------|-----|-------------------------Bad "var_declaration"
                                                         |     |                         Recognized:  var identifier ,
                                                         |     |                         Legal would have been:  "identifier"
                                                         |     |
****Error                                                |-----|-------------------------Bad "type_denoter"
                                                               |                         Recognized:  array [ <ordinal_type> ]
                                                               |                         Legal would have been:  "of"
                                                               |
****Error                                                      |-------------------------Bad "if_statement"
                                                                                         Recognized:  if <Boolean_expression>
                                                                                         Legal would have been:  "(","[",".","^",
                                                                                             "multiplying_operator","div","mod",
                                                                                             "and","adding_operator","or","=",
                                                                                             "relational_operator","in", or "then"
       18              begin
       19                x:=1
       20              end ;
       21
       22 begin  end.
```

```
        1
        2
        3  program sample3(input,output);
        4
        5  procedure p; begin
        6                 begin
        7                   top[indx]:=0;   count[indx]:=0;
        8                 end
        9                 for indx:=1 to norels do
                            |
****Error                   |----------------------------------------------------------------Bad "compound_statement"
                                                                                             Recognized:  begin <statement>
       10                   x:=1                                                             Legal would have been:  "end", or ";"
       11               end ;
       12
       13  procedure intlnklist(x, var y : integer);
                                     |
****Error                            |-------------------------------------------------------Bad "formal_parameter_list"
                                                                                             Recognized:  ( identifier ,
       14                      var q:integer;                                                Legal would have been:  "identifier"
       15                      begin x:=1 end;
       16
       17  procedure p; begin
       18                 if non push(i) then
                                 %%%%%%%
                                     |
****Error                            |-------------------------------------------------------Bad "if_statement"
                                                                                             Recognized:  if <Boolean_expression>
                                                                                             Legal would have been:  "(","[",".","^",
                                                                                                "multiplying_operator","div","mod",
                                                                                                "and","adding_operator","or","=",
       19                   x:=1                                                                "relational_operator","in", or "then"
       20               end ;
       21
       22  procedure p; begin
       23                 if x=1 then
       24                   begin
       25                     begin x:=1
       26                     end
       27                 else empty := true;
                             |
****Error                    |---------------------------------------------------------------Bad "compound_statement"
                                                                                             Recognized:  begin <statement>
       28                 x:=1                                                               Legal would have been:  "end", or ";"
       29               end ;
       30
       31  begin end.
```

```
        1
        2
        3  program sample4(input,output);
        4
        5  procedure p; begin
        6                 if not leftpush(listdata[p]
        7                    then writeln(' queue overflow.');
                                 |
****Error                        |-----------------------------------------------------------Bad "actual_parameter_list"
                                                                                             Recognized:  ( <expression>
                                                                                             Legal would have been:  "[",".","^",
                                                                                                "multiplying_operator","div","mod",
                                                                                                "and","adding_operator","or","=",
                                                                                                "relational_operator","in",")", or ","
        8               end ;
        9
       10  procedure p; begin
       11                 if(link[i]=0)and(rightpop(x)
       12                   then i:=x
                                |
****Error                       |------------------------------------------------------------Bad "factor"
                                                                                             Recognized:  ( <expression>
                                                                                             Legal would have been:
                                                                                                "multiplying_operator","div","mod",
                                                                                                "and","adding_ooerator","or","=",
                                                                                                "relational_operator","in", or ")"
       13               end ;
       14
       15  procedure p; begin
       16                 if count[listdata[sub] := 0
                                                  %%%%
                                                   |
****Error                                          |-----------------------------------------Bad "variable_access"
                                                                                             Recognized:  ( <expression>
                                                                                             Legal would have been:  "[",".","^",
                                                                                                "multiplying_operator","div","mod",
                                                                                                "and","addino_operator","or","=",
                                                                                                "relational_operator","in","]", or ","
       17                   then begin f := listdata[sub];
       18               end ; x:= 1; end
       19
       20  procedure hunter\input,output'?
                   |       %%%%%%%%%%%%%%%%
                   |        |             |
****Error          |--------|----------------------------------------------------------------Bad "proc/func_declaration"
                            |             |                                                  Recognized:  procedure identifier ;
                            |             |                                                     ... identifier ; <block>
                            |             |                                                  Lagal would have been:  ";"
                            |             |
****Error                   |-------------|--------------------------------------------------illegal character(s)
                            |             |
****Error                   |-------------|--------------------------------------------------Bad "proc/func_declaration"
                                          |                                                  Recognized:  orocedure identifier ;
                                          |                                                     ... <block> procedure identifier
                                          |                                                  Legal would have been:  "(", or ";"
                                          |
****Error                                 |--------------------------------------------------unclosed string quote at and of line
       21                   ver q:integer;
                            %%%%%
       22                   begin x:=1 end ;
       23
       24  begin end.
```

```
        1
        2
        3 program sample5(input,output);
        4
        5 procedure p; begin  for i  1 to 6 do
                                     $
                                     |
****Error                            |----------------------------------------------------Bad "for_statement"
                                                                                          Recognized:  for identifier
                                                                                          Legal would have been:  ":="
        6                x:=1
        7              end;
        8
        9 procedure p; begin
       10                writeln(no. of primes less than@,a[i],@ is @,numb);
                                     $$$$$$$$$$$$$$$$$$$$      $$$$$$
                                      |                        |
****Error                             |------------------------|--------------------------Bad "variable_access"
                                                               |                          Recognized:   .
                                                               |                          Legal would have been:  "identifier"
                                                               |
****Error                                                      |--------------------------Bad "write_parameter_list"
                                                                                          Recognized:  ( <expression> ,
                                                                                             <expression> ,
                                                                                          Legal would have been:
                                                                                             "adding_operator","unsigned_integer",
                                                                                             "unsigned_real","character_string",
                                                                                             "nil","identifier","[","(", or "not"
       11               x:=1 end ;
       12 procedure p;  const pi := 3.14159;
                                 $$$$$$$$$$$
                                  |
****Error                         |-------------------------------------------------------Bad "const_declaration"
                                                                                          Recognized:  const identifier
                                                                                          Legal would have been:  "="
       13               begin end;
       14
       15 procedure p;
       16   function power(m,n: integer): integer;
       17     begin
       18       if n<= then power := else power := m*power(m,m-1)
                           |             |
****Error                  |-------------|------------------------------------------------Bad "Boolean_expression"
                                         |                                                Recognized:  <simple_expression>
                                         |                                                     relational_operator
                                         |                                                Legal would have been:
                                         |                                                   "adding_operator","unsigned_integer",
                                         |                                                   "unsigned_real","character_string",
                                         |                                                   "nil","identifier","[","(", or "not"
                                         |
****Error                                |------------------------------------------------Bad "statement"
                                                                                          Recognized:  identifier :=
                                                                                          Legal would have been:
                                                                                             "adding_operator","unsigned_integer",
                                                                                             "unsigned_real","character_string",
                                                                                             "nil","identifier","[","(", or "not"
       19     end;
       20   begin  end;
       21
       22 begin   end.
       1
       2
       3 program sample6(input,output);
       4
       5 procedure p;
       6            var x,m,i integer; fx,sx,lx real;
                              $$$$$$$           $$$$
                                    |              |
****Error                           |--------------|-------------------------------Bad "var_declaration"
                                                   |                               Recognized:  var identifier ,
                                                   |                                  identifier , identifier
                                                   |                               Legal would have been:  ":", or ","
                                                   |
****Error                                          |-------------------------------Bad "var_declaration"
                                                                                   Recognized:  var identifier , ...
                                                                                      identifier , identifier
                                                                                   Legal would have been:  ":", or ","

       7            begin x:=1 end;
       8
       9 procedure p;
      10            begin label 1;
                              |
****Error                     |----------------------------------------------------Bad "compound_statement"
                                                                                   Recognized:  begin
                                                                                   Legal would have been:
                                                                                      "unsigned_integer","identifier","goto",
                                                                                      "write","writeln","begin","if","case",
                                                                                      "repeat","while","for","with","end",
                                                                                       or ";"

      11              read(x);  x:=1 end;
                      $$$$$$$

      12 procedure p; begin
      13              1: prcount := prcount;
      14
      15 procedure pr; begin x:=1 end ;
                 |
****Error        |-----------------------------------------------------------------Bad "compound_statement"
                                                                                   Recognized:  begin <statement> ;
                                                                                   Legal would have been:
                                                                                      "unsigned_integer","identifier","goto",
                                                                                      "write","writeln","begin","if","case",
                                                                                      "repeat","while","for","with","end",
                                                                                       or ";"

      16              var  fact, fact2: real ;
                      $$$$$$$$$$$$$$$$$$$$$$$
                        |
****Error               |----------------------------------------------------------Bad "block"
                                                                                   Recognized:  <proc/func_declaration>
                                                                                   Legal would have been:  "procedure",
                                                                                      "function", or "begin"

      17              begin x:=1 end;
                                   $

      18
      19 procedure p;
      20             type word = array (1..10) of char;
                                       $$$$$$$
                                       |
****Error                              |-------------------------------------------Bad "type_denoter"
                                                                                   Recognized:  array
                                                                                   Legal would have been:  "["

      21             begin    end;
      22 begin  end.
```

```
        2 program sample7(input,output);
        3 procedure insertlegal; var q r: legalptr;
                                       $
                                       |
****Error                              |--------------------------------------------------Bad "var_declaration"
                                                                                          Recognized:  ver identifier
        4 begin  new q); q^..boxnum:= pos;                                                Legal would have been:  ";", or ","
                     $$    $$$$$$$$
                     |     |
****Error            |-----|--------------------------------------------------------------Bad "compound_stetement"
                           |                                                              Recognized:  begin <statement>
                           |                                                              Legal would have been:  "(","[",".","^",
                           |                                                                  ":=","end", or ";"
                           |
****Error                  |--------------------------------------------------------------Bad "statement"
                                                                                          Recognized:  identifier
                                                                                              <variable_access>
                                                                                          Legal would have been:  "[",".","^",
                                                                                               or ":="
        5   q^.next:= nil; if p:= nil then p:= q
                                $$$$$$
                                |
****Error                       |---------------------------------------------------------Bad "if_statement"
                                                                                          Recognized:  if <Booleen_expression>
                                                                                          Legal would heve been:  "(","[",".","^",
                                                                                              "multiplying_operator","div","mod",
                                                                                              "and","adding_operetor","or","=",
        6   else else begin r:= p;                                                        "relational_operetor","in", or "then"
                 $$$$
                 |
****Error        |------------------------------------------------------------------------Bad "compound_statement"
                                                                                          Recognized:  begin <statement> junk
                                                                                              ... <statement> ; <stetement>
                                                                                          Legel would have been:
                                                                                              "unsigned_integer","identifiar","goto",
                                                                                              "write","writeln","bagin","if","cese",
                                                                                              "repeat","wnila","for","with","end",
                                                                                               or ";"
        7         while r^.next <> nil do r = r^.next;
                                             $$$$$$$$$$
                                             |
****Error                                    |--------------------------------------------Bad "compound_statement"
                                                                                          Recognized:  begin <statement> ;
                                                                                              <statament>
                                                                                          Legal would have been:  "(","[",".","^",
        8         r^.next:= q end end end                                                     ":=","end", or ";"
                                       $$$
                                       |
****Error                              |--------------------------------------------------Bad "proc/func_declaretion"
                                                                                          Recognized:  procedure identifier ;
                                                                                              <block>
        9 end; begin end;                                                                 Legal would have been:  ";"
          $$$           $
                        |
****Error               |-----------------------------------------------------------------Bad "Program"
                                                                                          Recognized:  program identifier ( ...
                                                                                              ) ; <block>
       10 begin   end.                                                                    Legal would have been:  "."
```

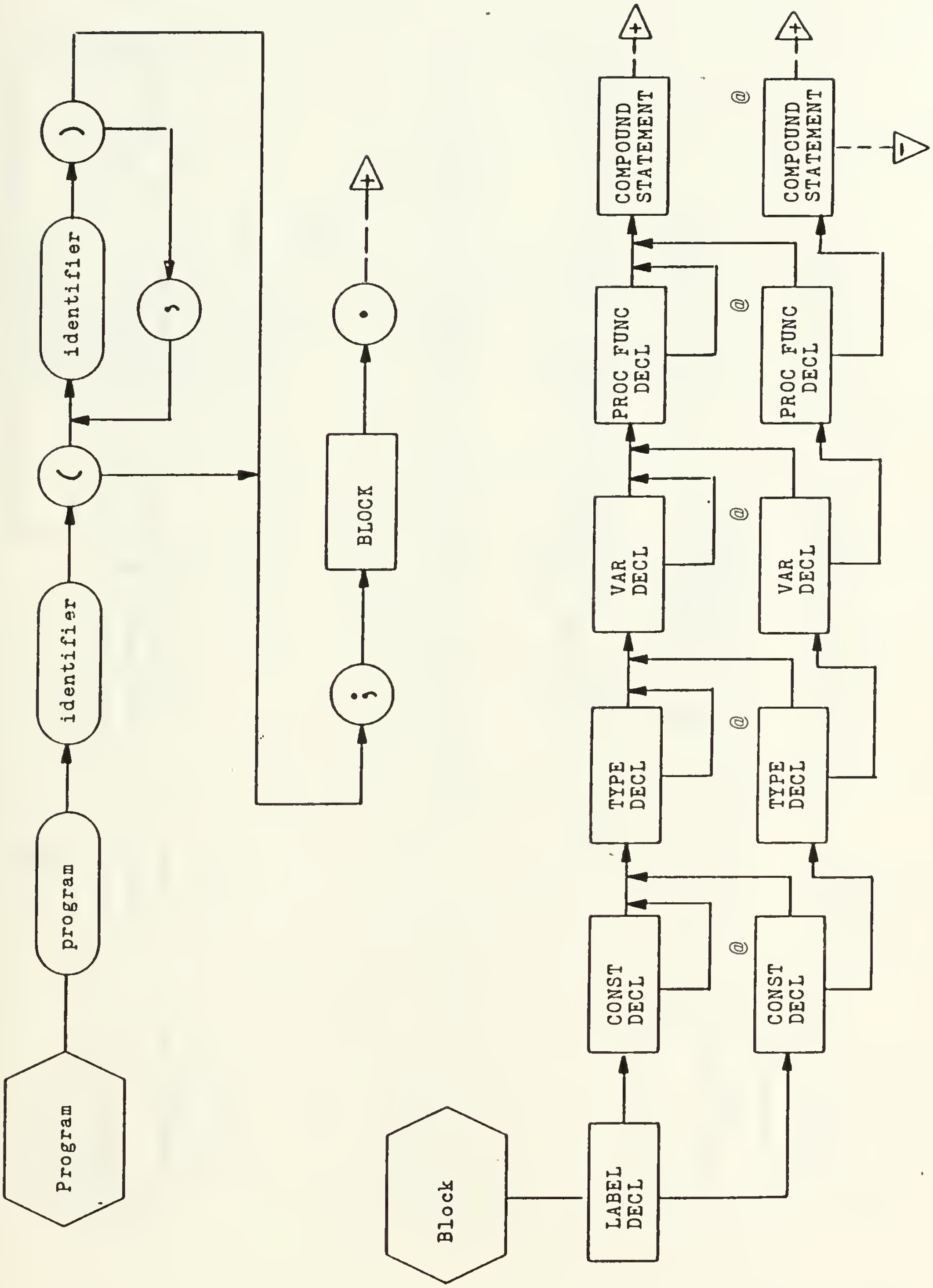
APPENDIX B: TRANSITION DIAGRAMS

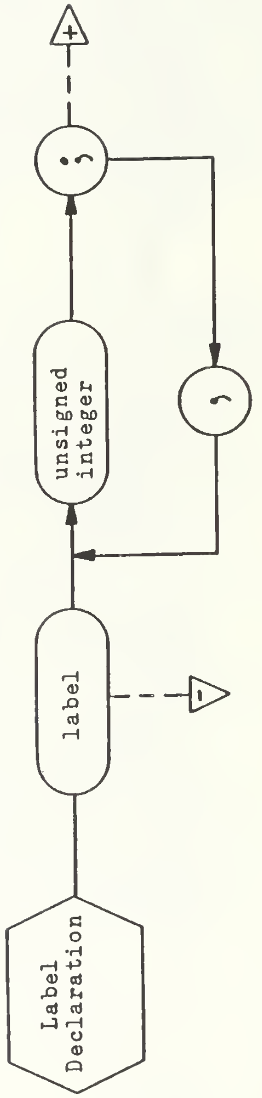
The following are the transition diagrams which are traversed by the parser during syntactic analysis. As discussed in Chapters Two and Three, these diagrams are derived from the syntax charts, but have been modified to provide *unique* true and false exits for each syntactic unit. The table below illustrates the notation used in the transition diagrams. Exit arrow convention concerns the *initial* direction of the line from a box. Notice that while true exits are normally shown to the right, left is also used here due to space and readability considerations.

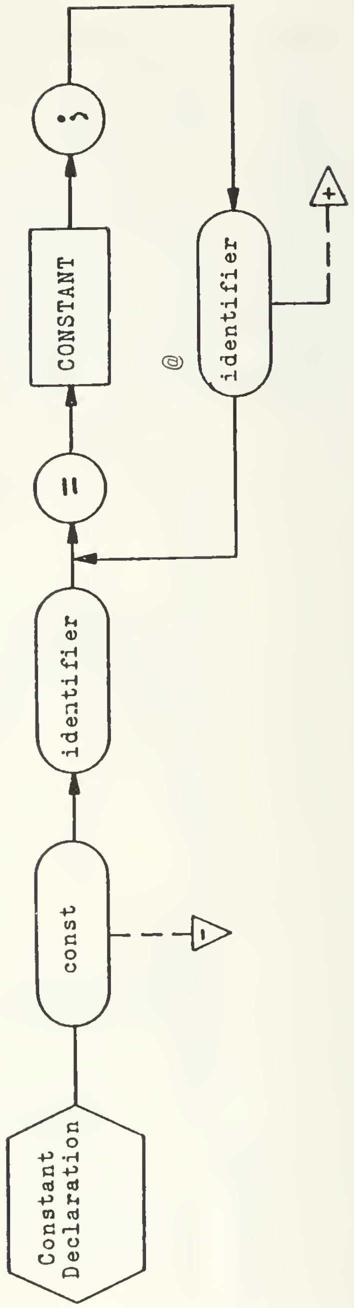
Diagram Symbology			
Symbol	Meaning	Symbol	Meaning
——> <——	true exit	@	Note
↓	false exit	⬡	header box
——▷ (+)	return true	▭	nonterminal box
↓ ▽ (−)	return false	○ ⬭	lexeme boxes

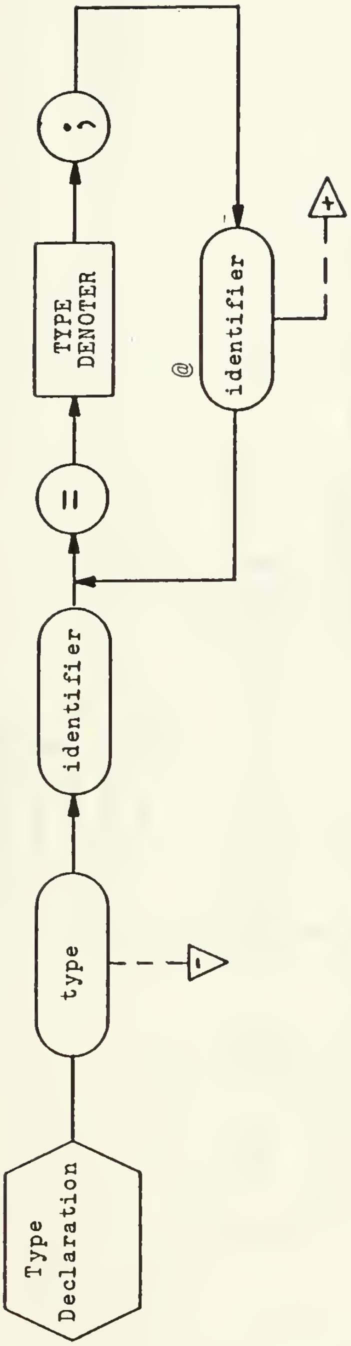
Note:

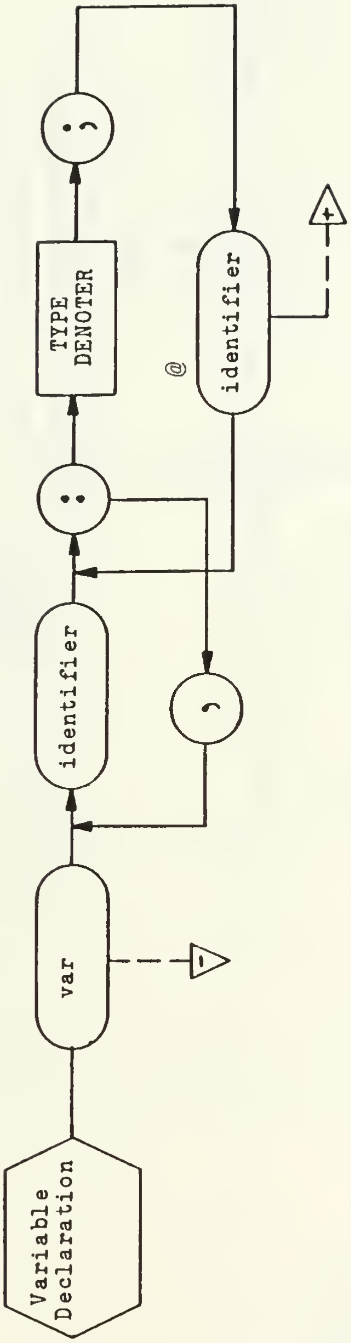
This symbol appears next to those boxes which have been added as a result of the modifications discussed in Chapter Three.

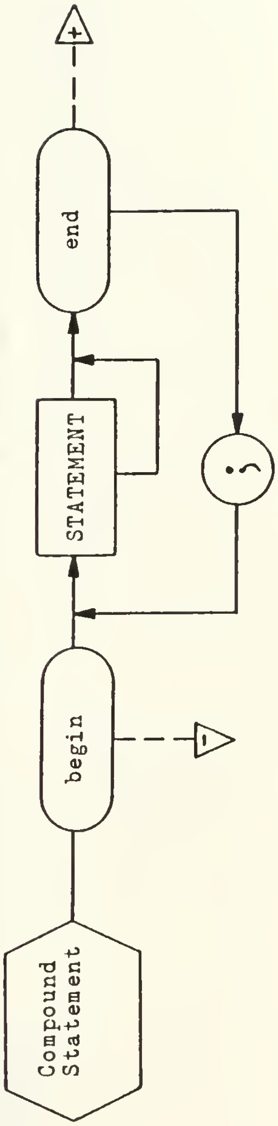

Program
program
identifier
(
identifier
;
)
;
BLOCK
.
+
Block
LABEL DECL
CONST DECL
TYPE DECL
VAR DECL
PROC FUNC DECL
COMPOUND STATEMENT
+
@
@
@
@
@
CONST DECL
TYPE DECL
VAR DECL
PROC FUNC DECL
COMPCUND STATEMENT
+
-

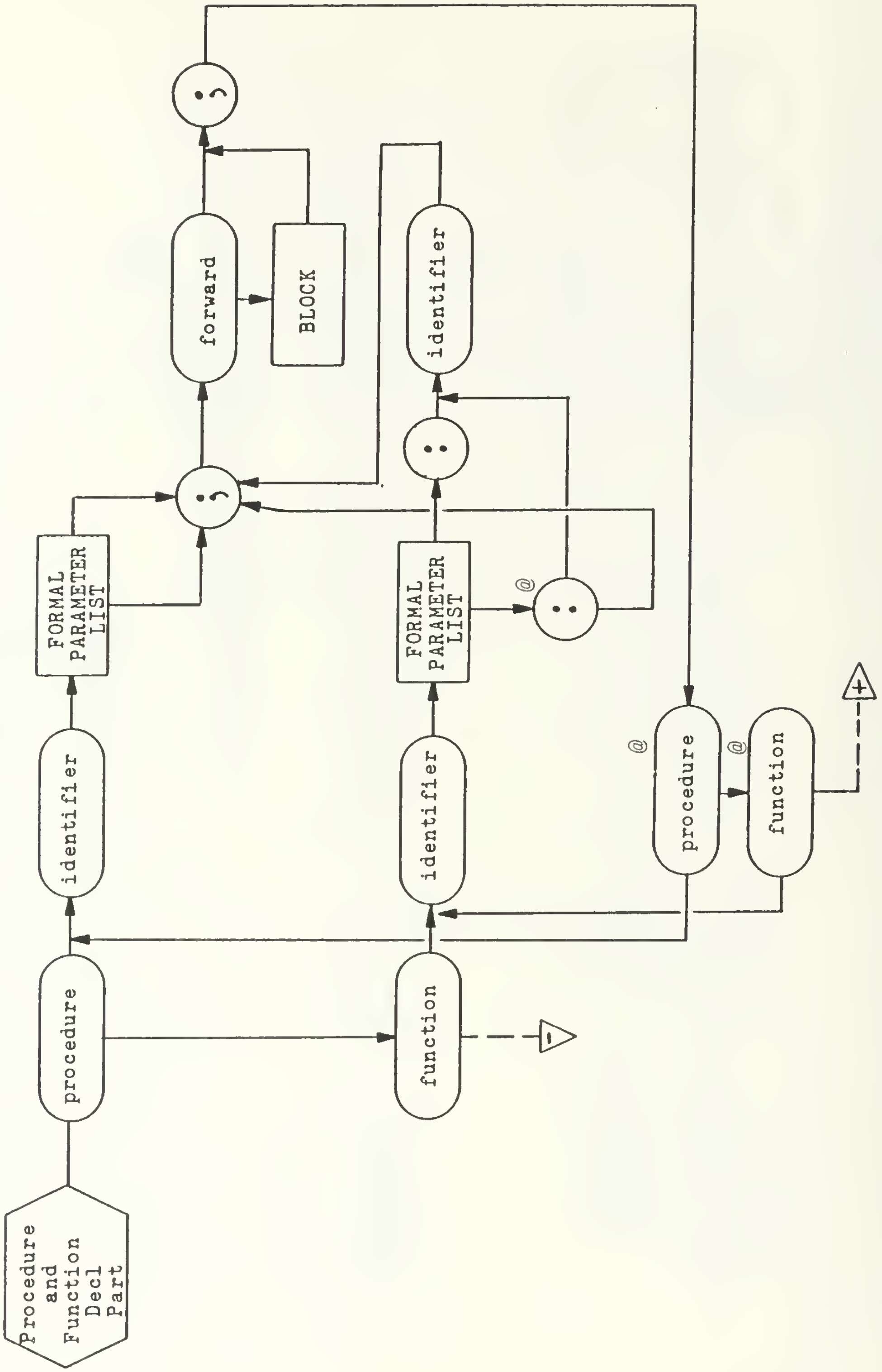

Label Declaration
label
unsigned integer
;
,
+
-

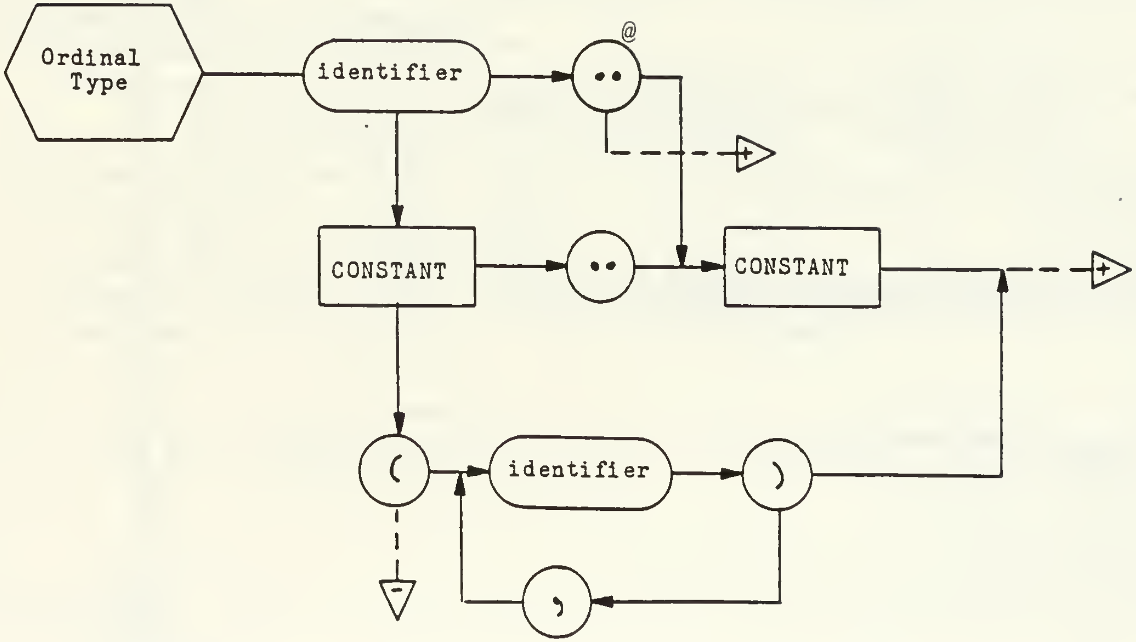

Constant Declaration
const
identifier
=
CONSTANT
;
@
identifier
+
-

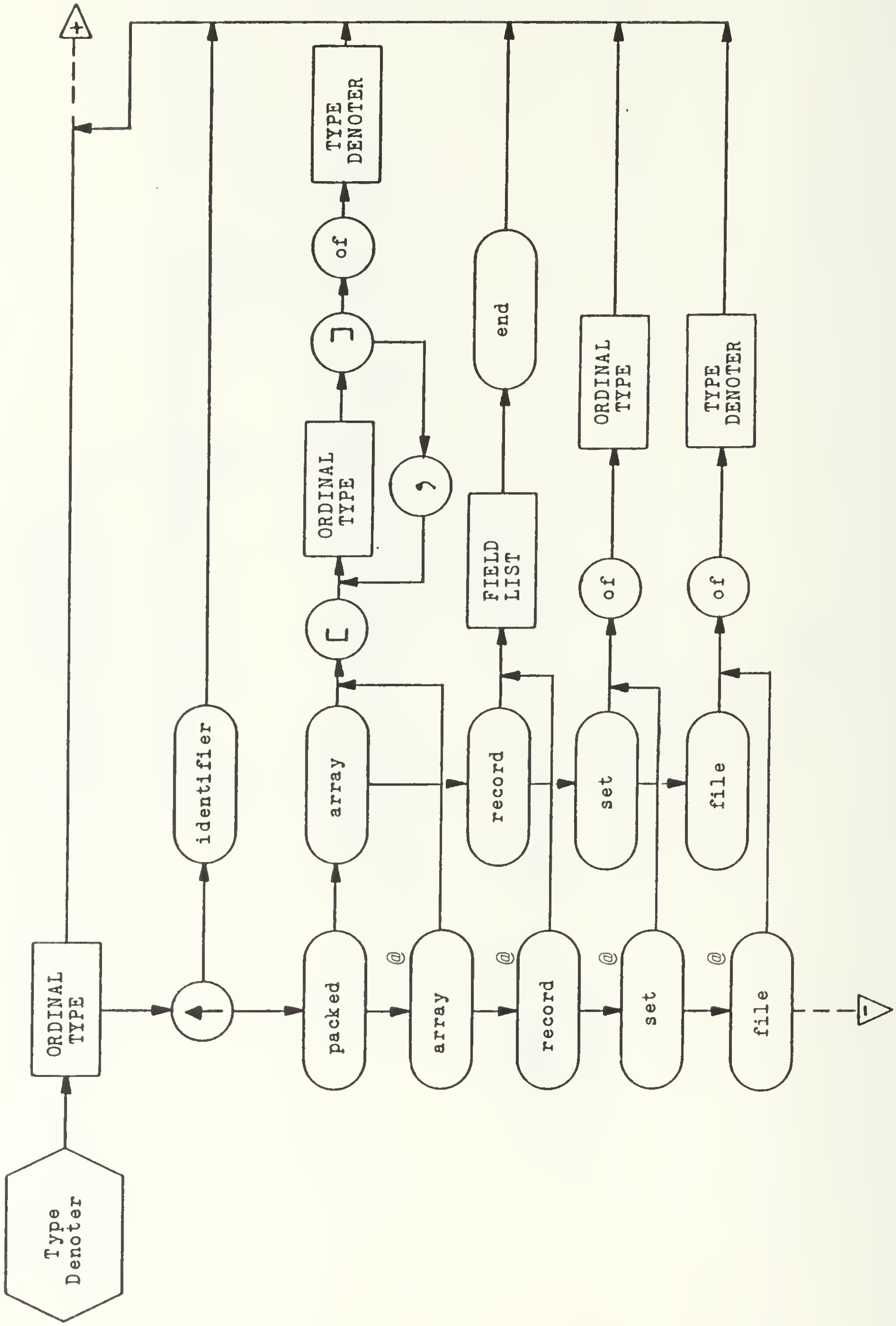

Type Declaration
type
identifier
=
TYPE DENOTER
;
@
identifier
-
+

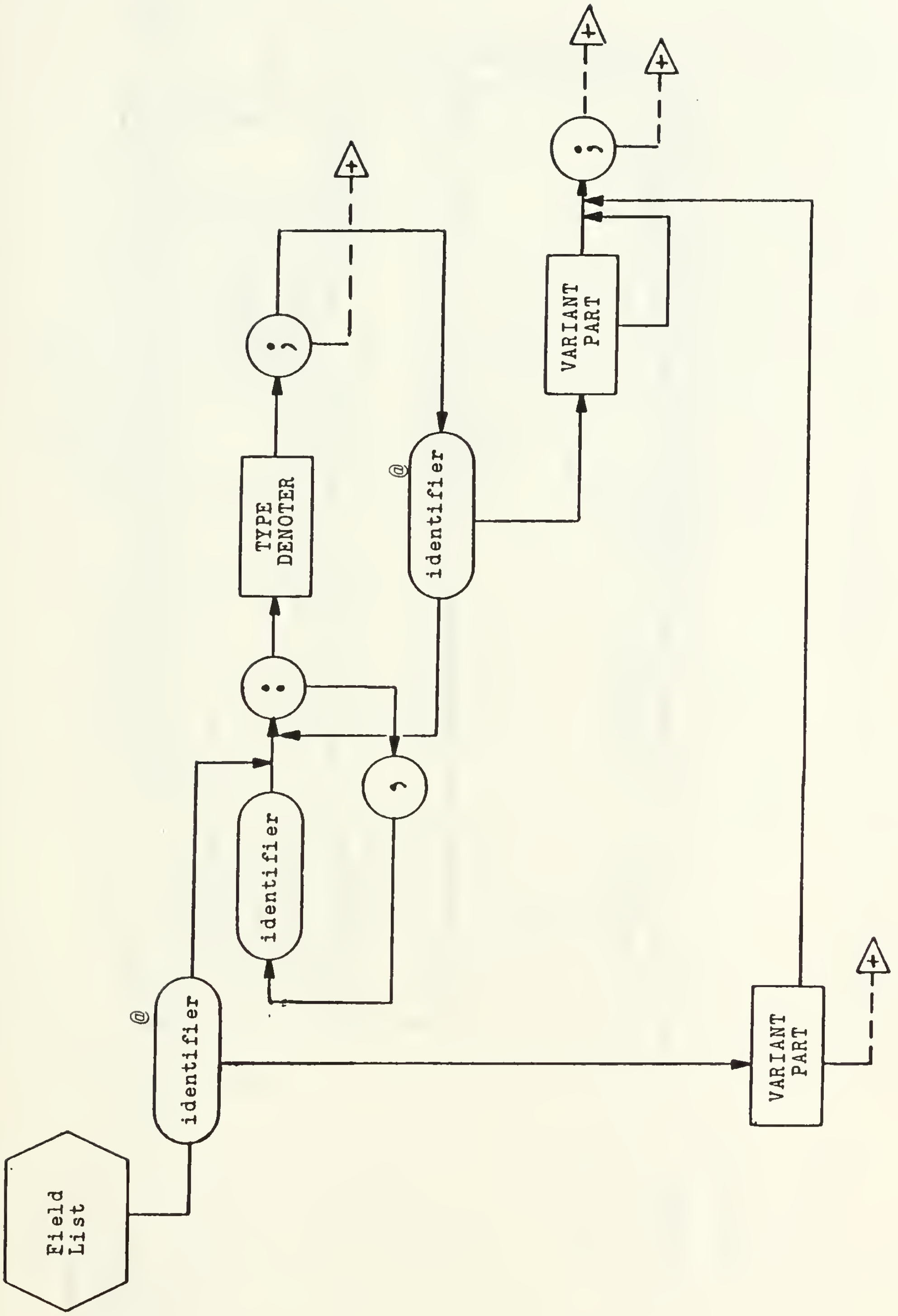

Variable Declaration
var
identifier
,
:
TYPE DENOTER
;
@
identifier
-
+

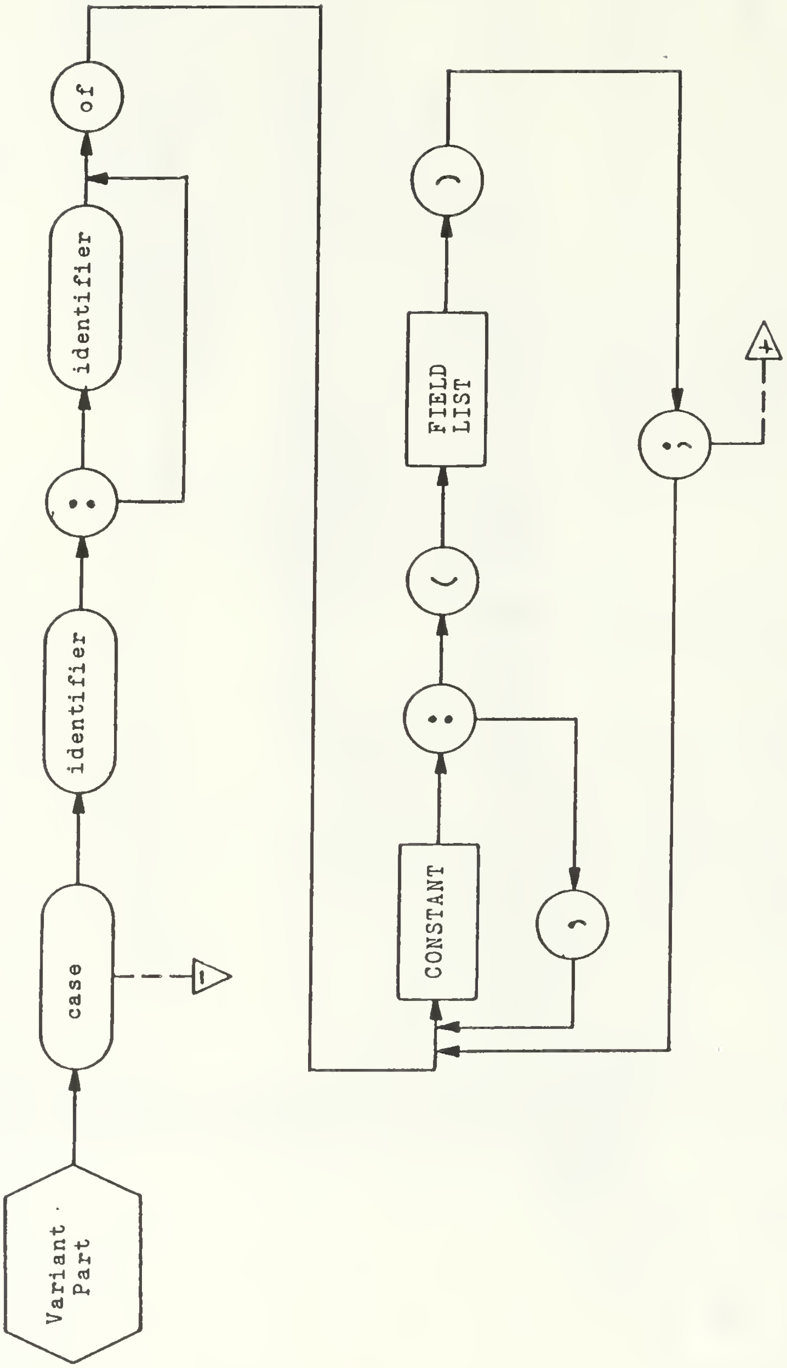

Compound Statement
begin
STATEMENT
;
end
-
+

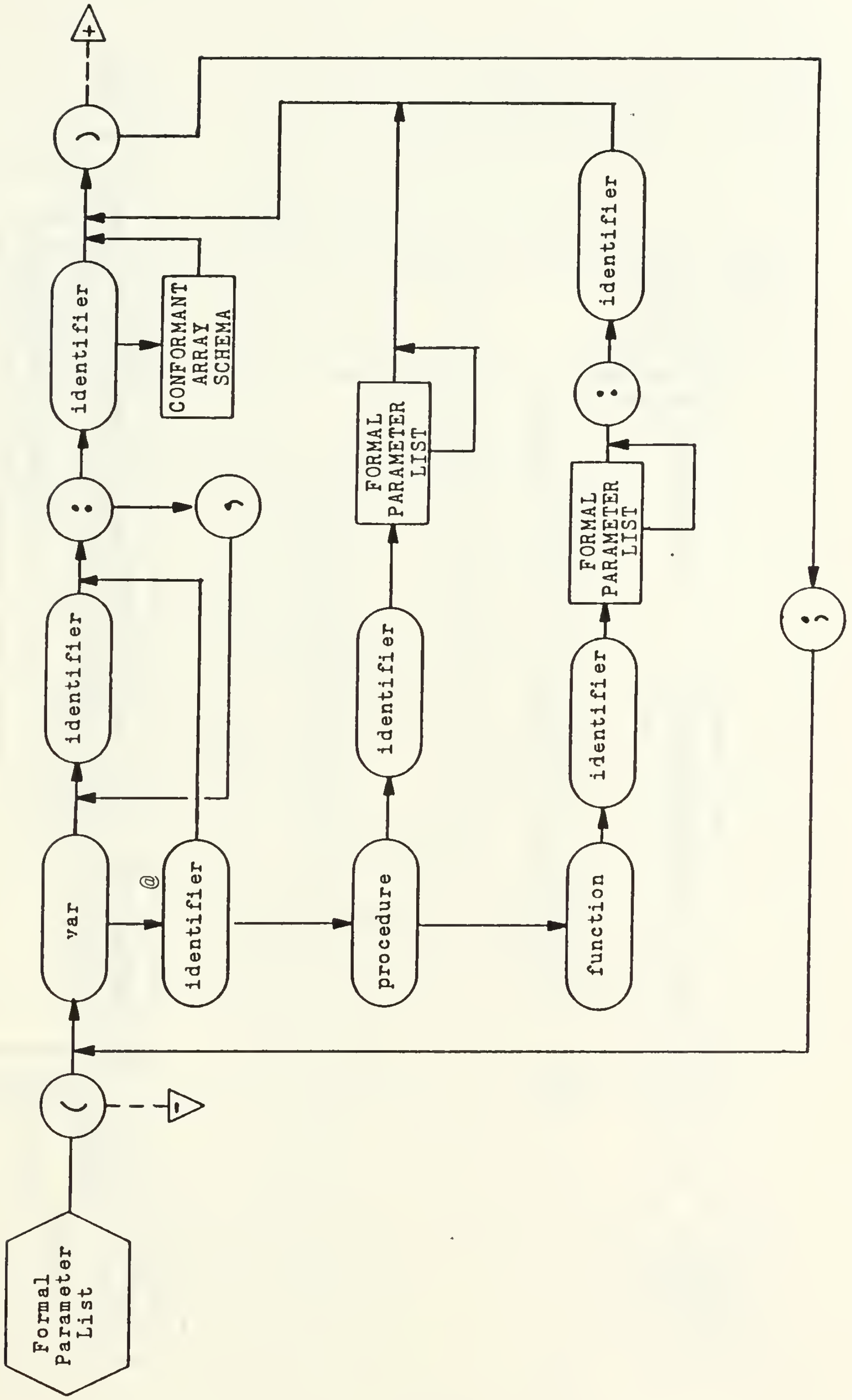

Procedure and Function Decl Part
procedure
function
identifier
identifier
FORMAL PARAMETER LIST
FORMAL PARAMETER LIST
;
:
@
:
forward
BLOCK
identifier
;
@
procedure
@
function
+
-

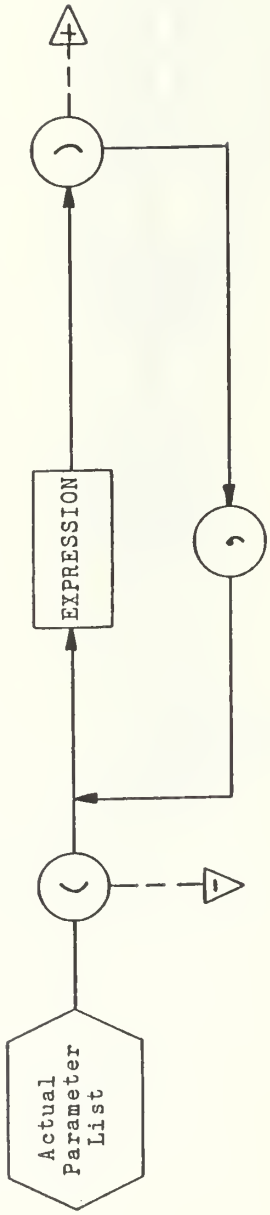

Ordinal Type
identifier
@
••
+
CONSTANT
••
CONSTANT
+
(
-
identifier
)
,

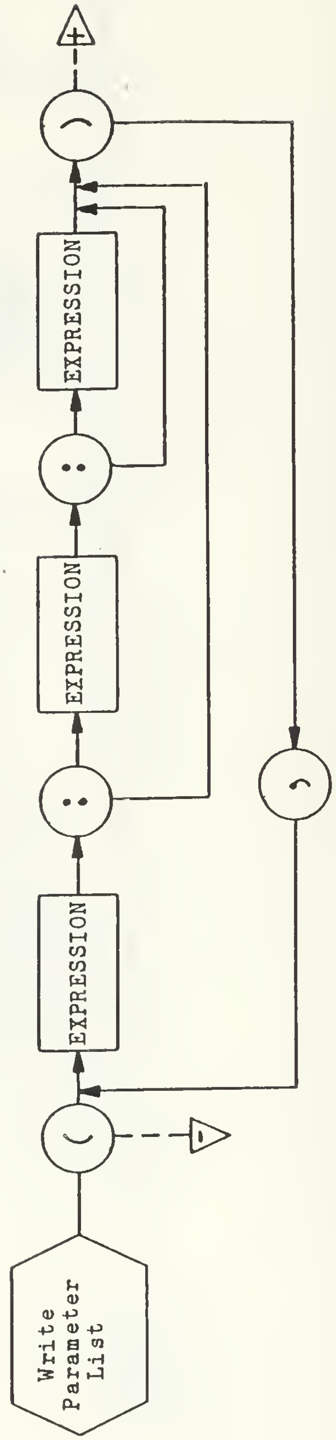

Type Denoter
ORDINAL TYPE
+
↑
identifier
packed
array
@
array
[
ORDINAL TYPE
,
]
of
TYPE DENOTER
@
record
record
FIELD LIST
end
@
set
set
of
ORDINAL TYPE
@
file
file
of
TYPE DENOTER
-

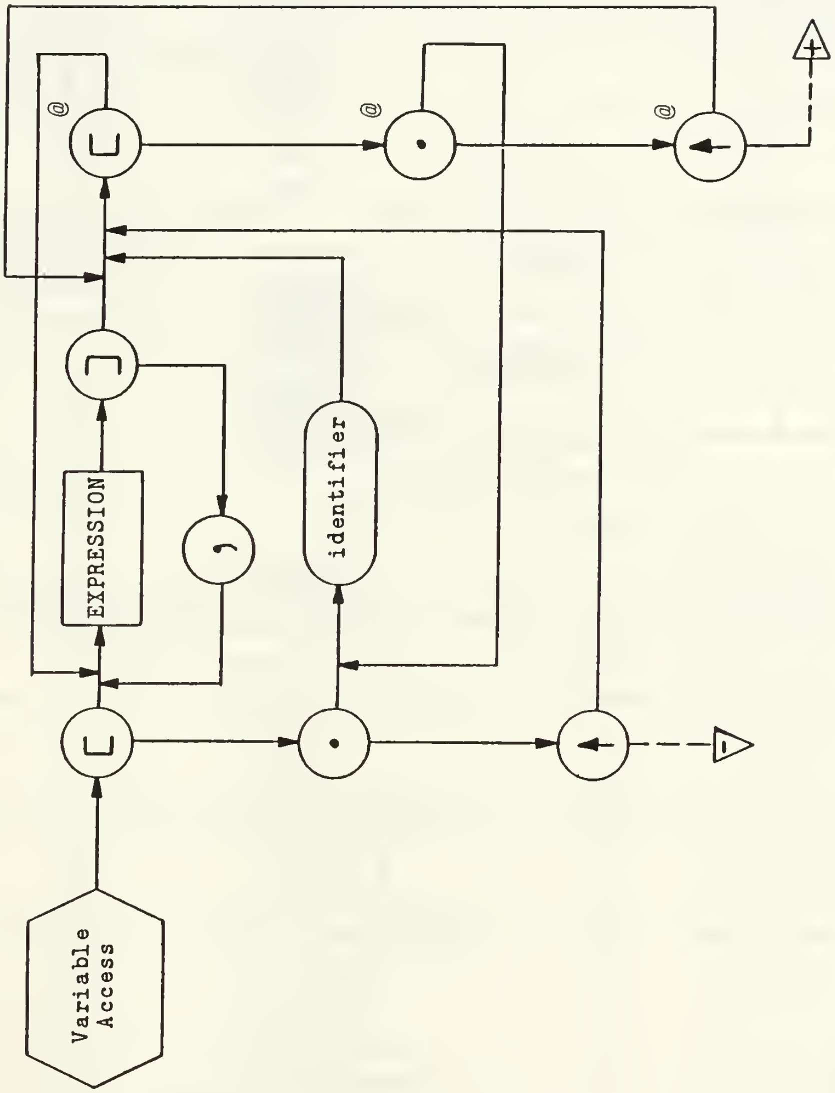

Field List
@
identifier
identifier
,
:
TYPE DENOTER
;
+
@
identifier
VARIANT PART
;
+
+
VARIANT PART
+

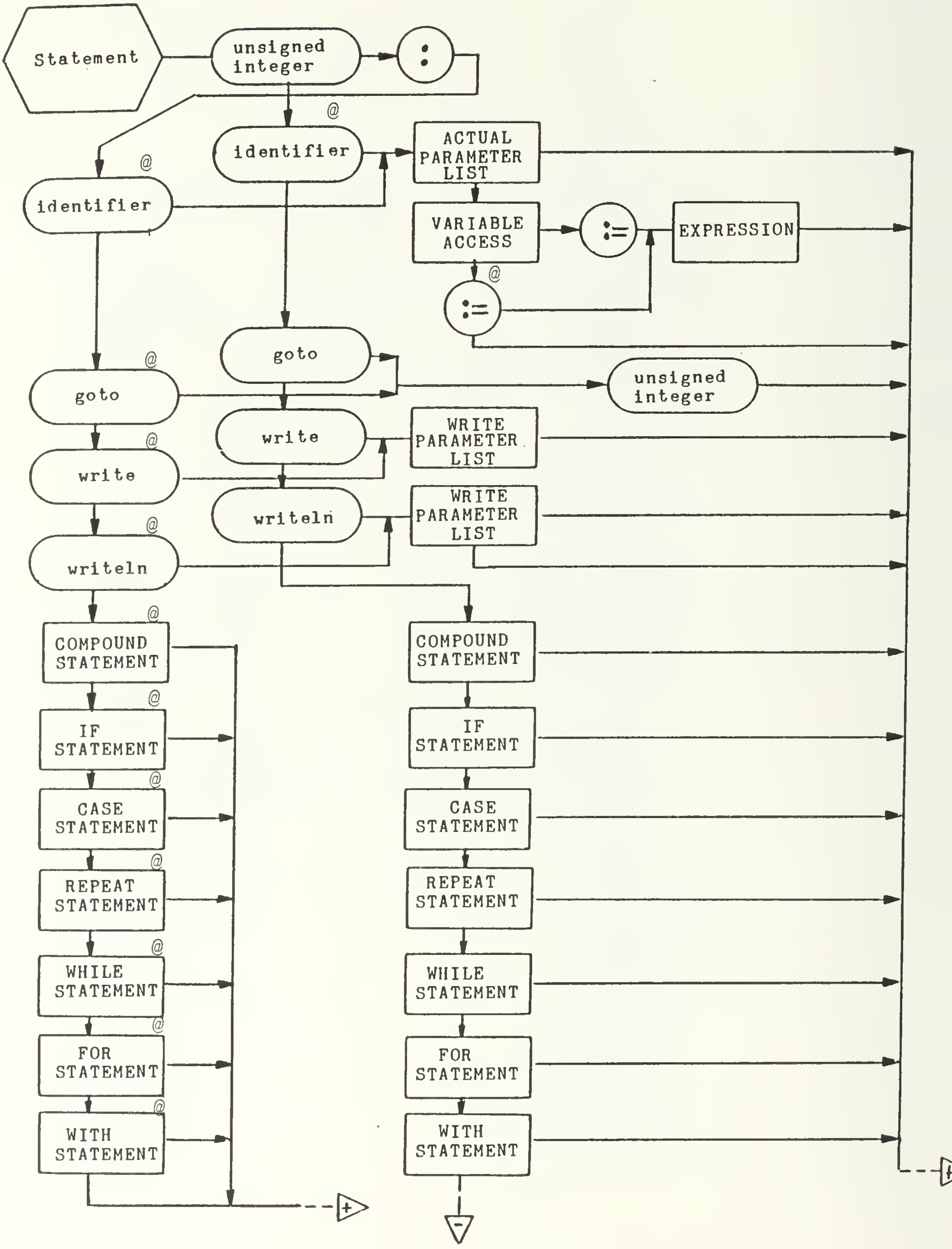

Variant Part
case
-
identifier
:
identifier
of
CONSTANT
:
(
FIELD LIST
)
,
;
+

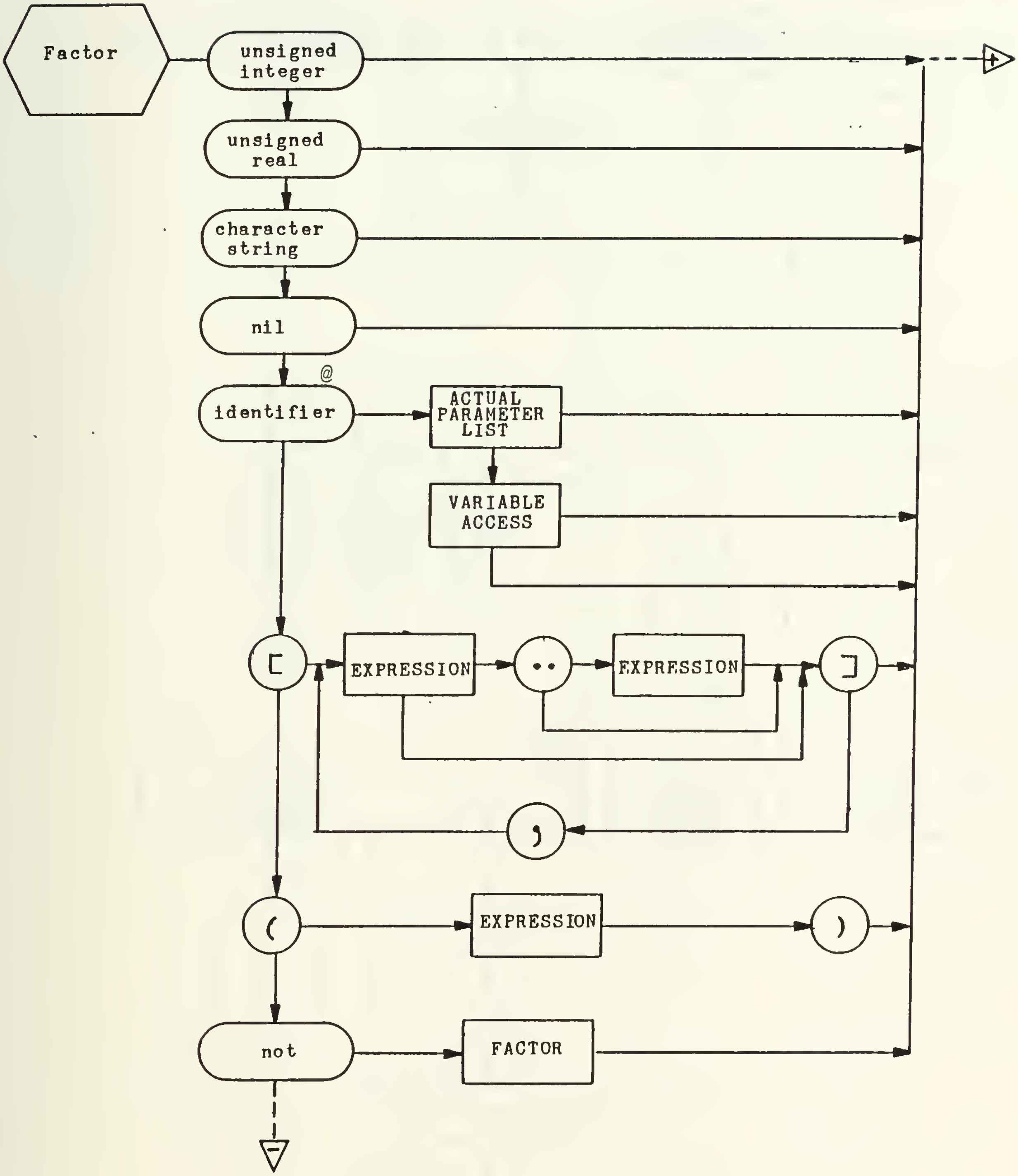

Formal Parameter List
(
-
var
@
identifier
procedure
function
identifier
identifier
identifier
:
;
identifier
CONFORMANT ARRAY SCHEMA
FORMAL PARAMETER LIST
FORMAL PARAMETER LIST
:
identifier
;
)
+

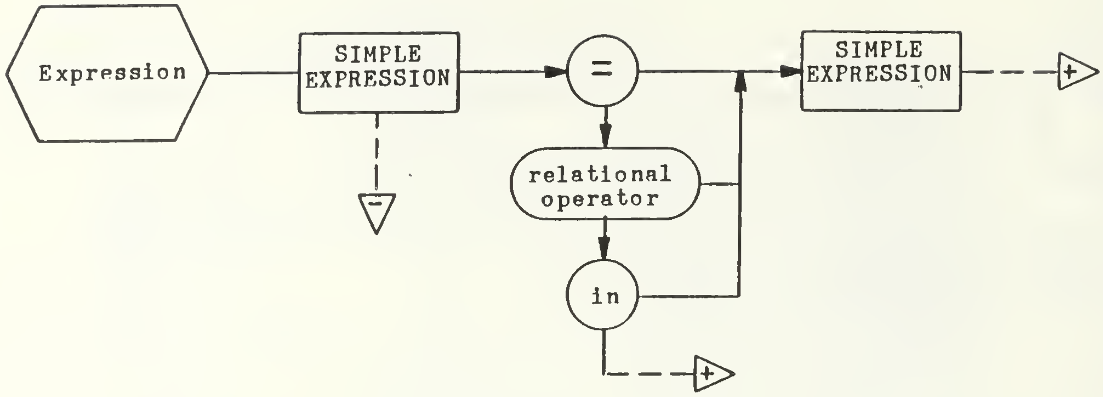

Actual Parameter List
(
EXPRESSION
,
)
+
-

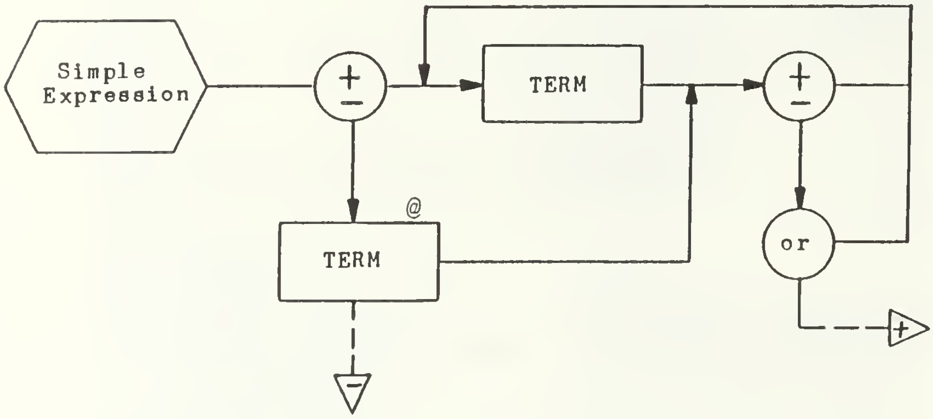

Write Parameter List
(
EXPRESSION
:
EXPRESSION
:
EXPRESSION
,
)
+
-

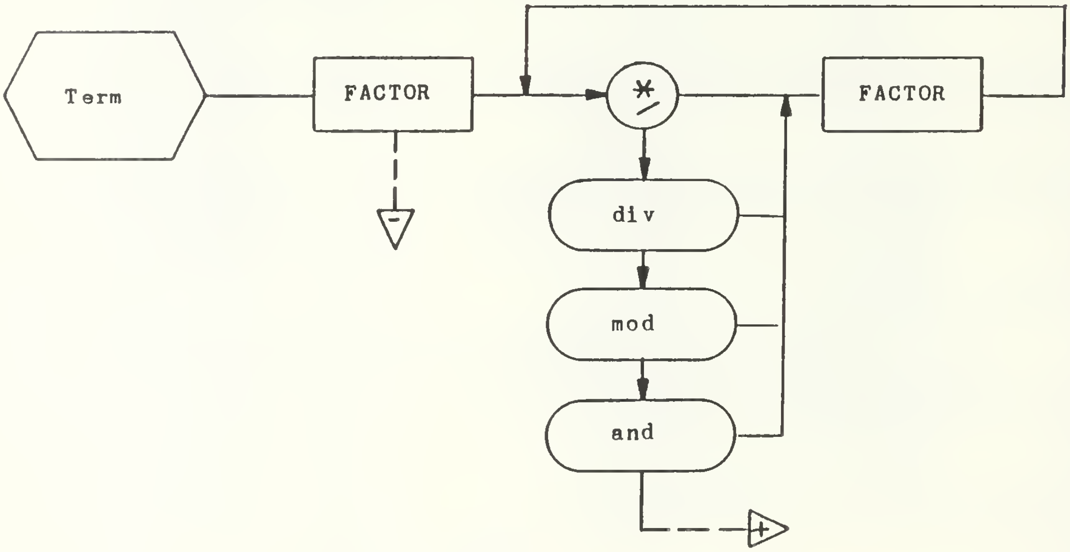

Variable Access
EXPRESSION
identifier
@
@
@

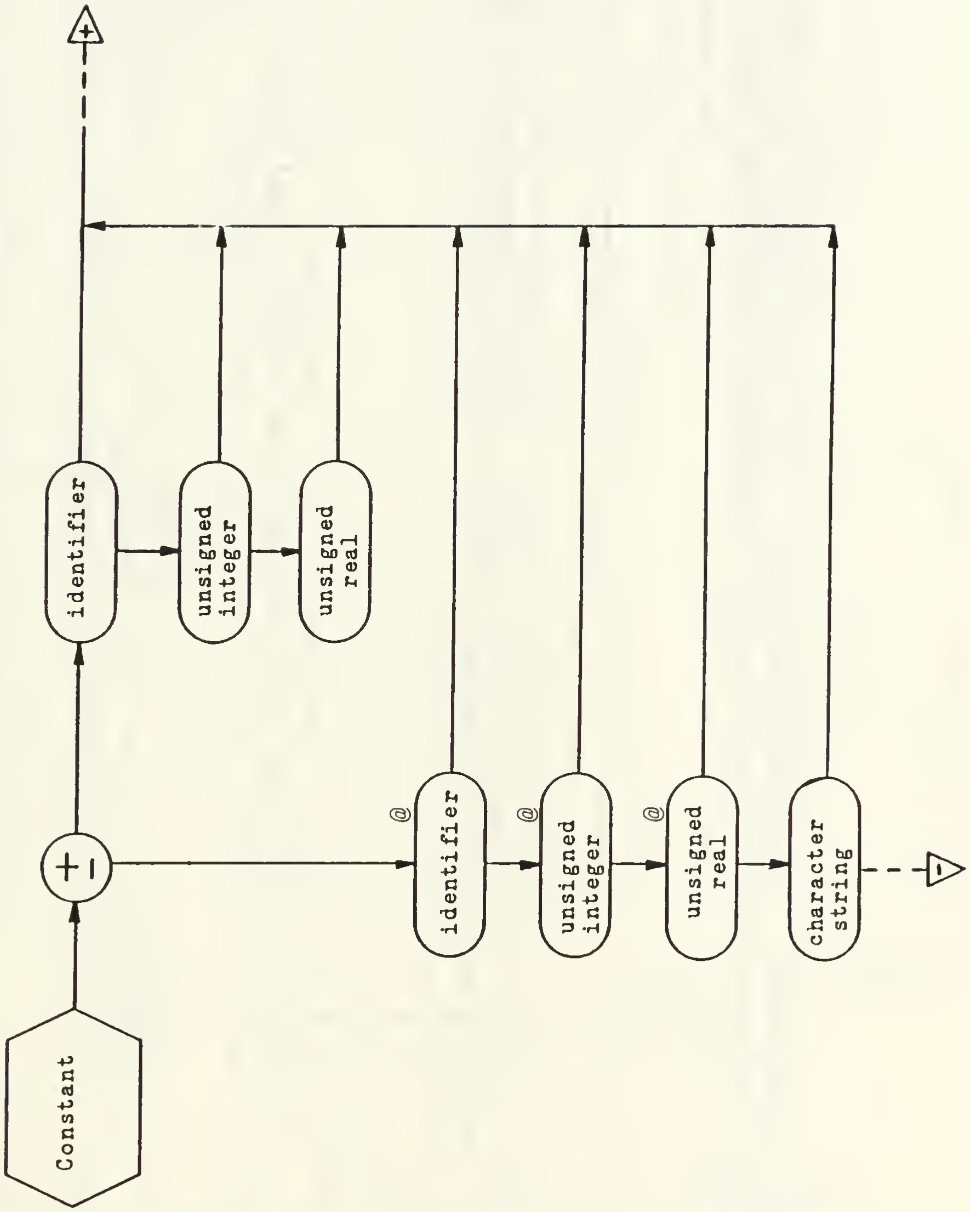

Statement
unsigned integer
:
@
identifier
ACTUAL PARAMETER LIST
identifier
VARIABLE ACCESS
:=
EXPRESSION
:=
goto
goto
unsigned integer
write
WRITE PARAMETER LIST
write
writeln
WRITE PARAMETER LIST
writeln
COMPOUND STATEMENT
COMPOUND STATEMENT
IF STATEMENT
IF STATEMENT
CASE STATEMENT
CASE STATEMENT
REPEAT STATEMENT
REPEAT STATEMENT
WHILE STATEMENT
WHILE STATEMENT
FOR STATEMENT
FOR STATEMENT
WITH STATEMENT
WITH STATEMENT
+
-
+

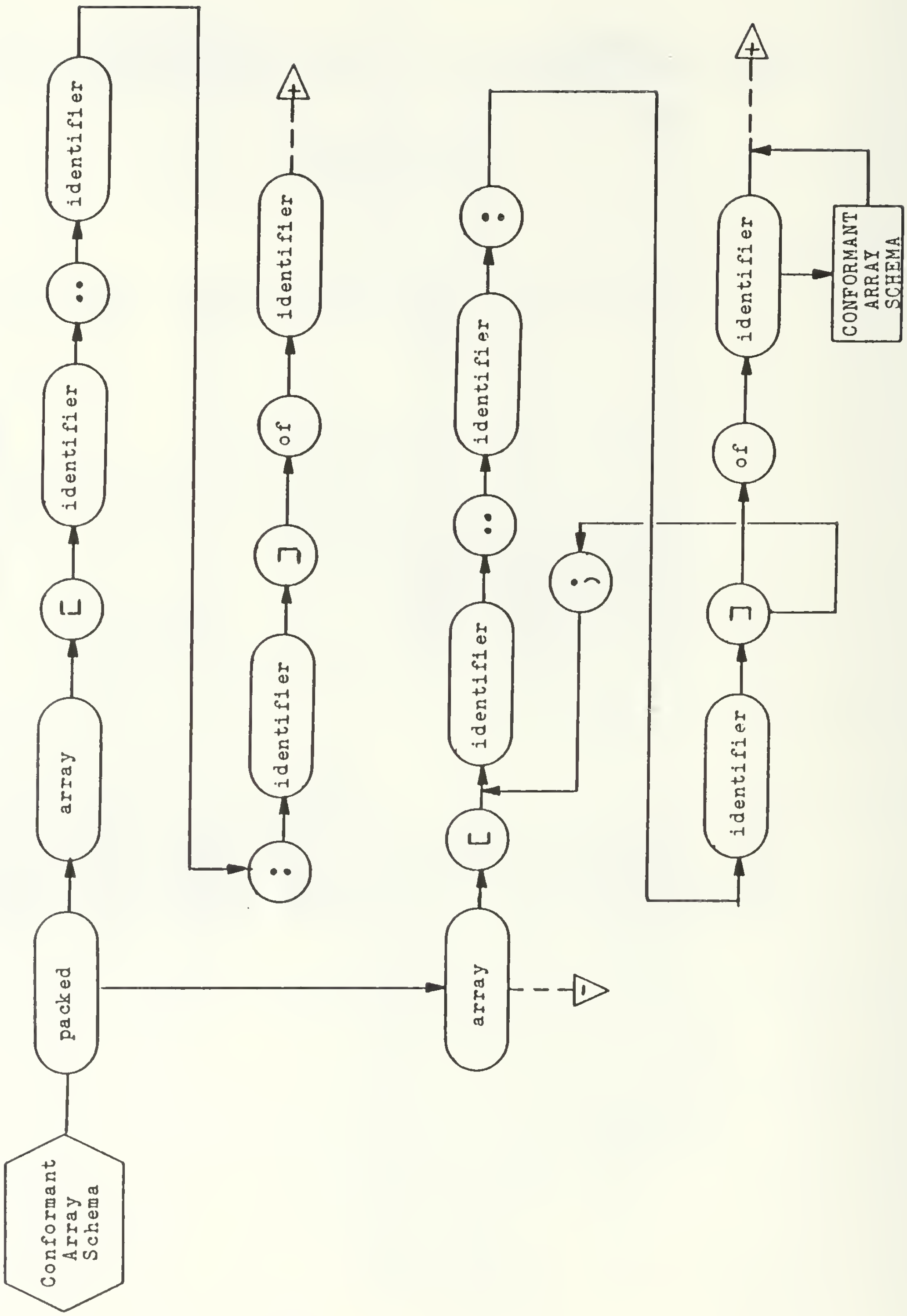

Factor
unsigned integer
unsigned real
character string
nil
@
identifier
ACTUAL PARAMETER LIST
VARIABLE ACCESS
[
EXPRESSION
..
EXPRESSION
]
,
(
EXPRESSION
)
not
FACTOR
+
−

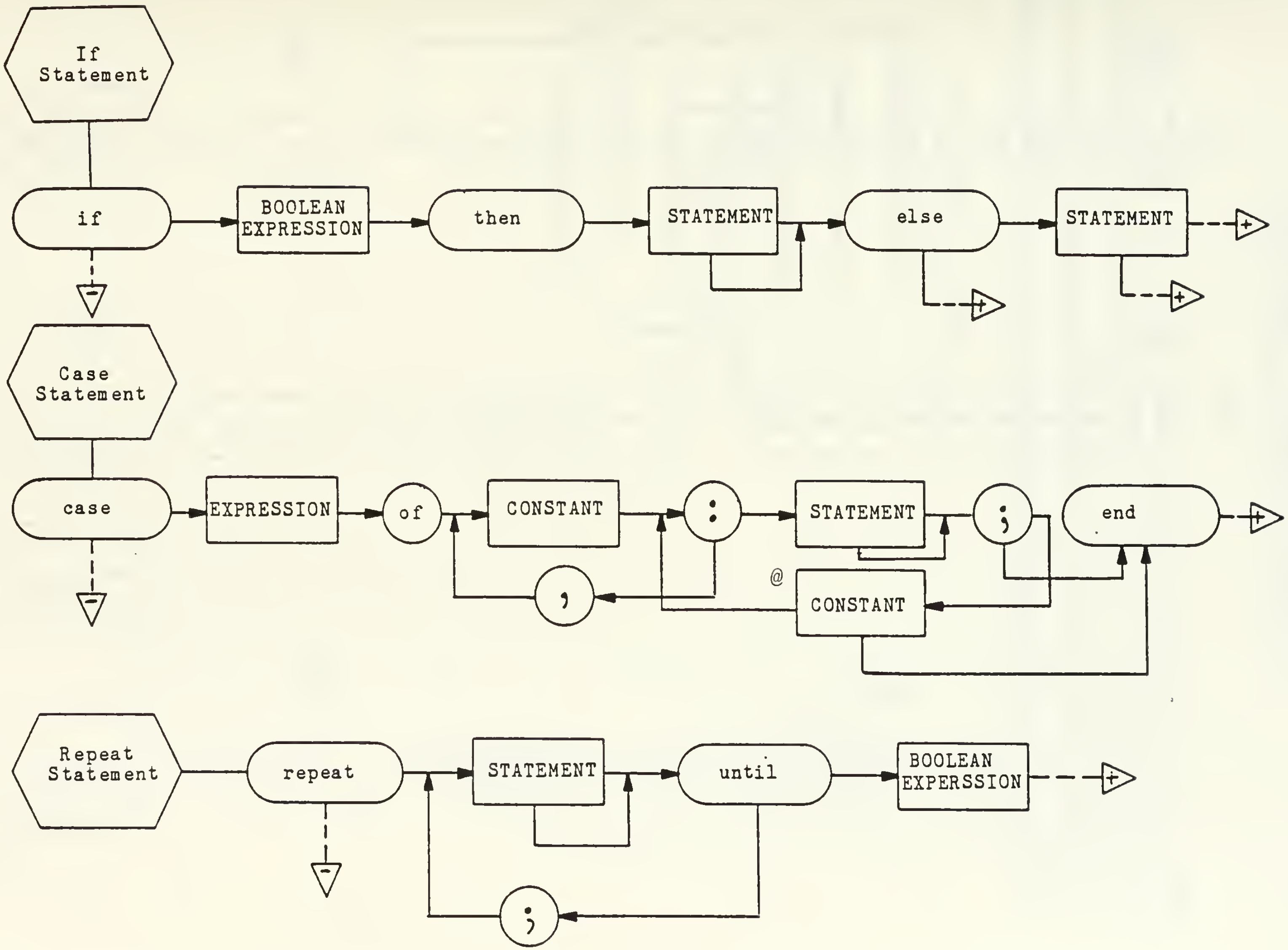

Expression
SIMPLE EXPRESSION
=
relational operator
in
SIMPLE EXPRESSION
+
−
+

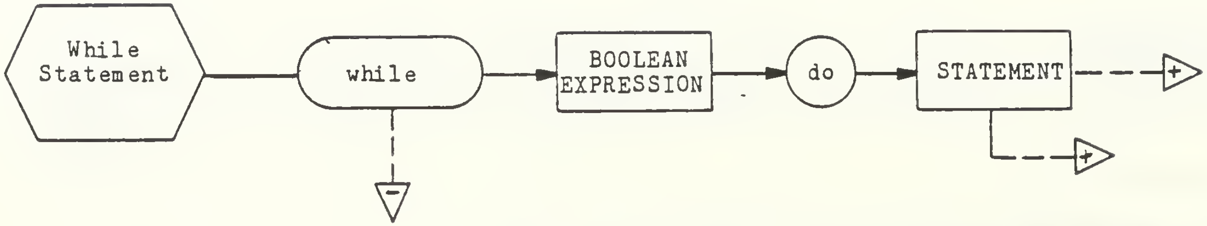

Simple Expression
+
−
TERM
+
−
or
@
TERM
+
−

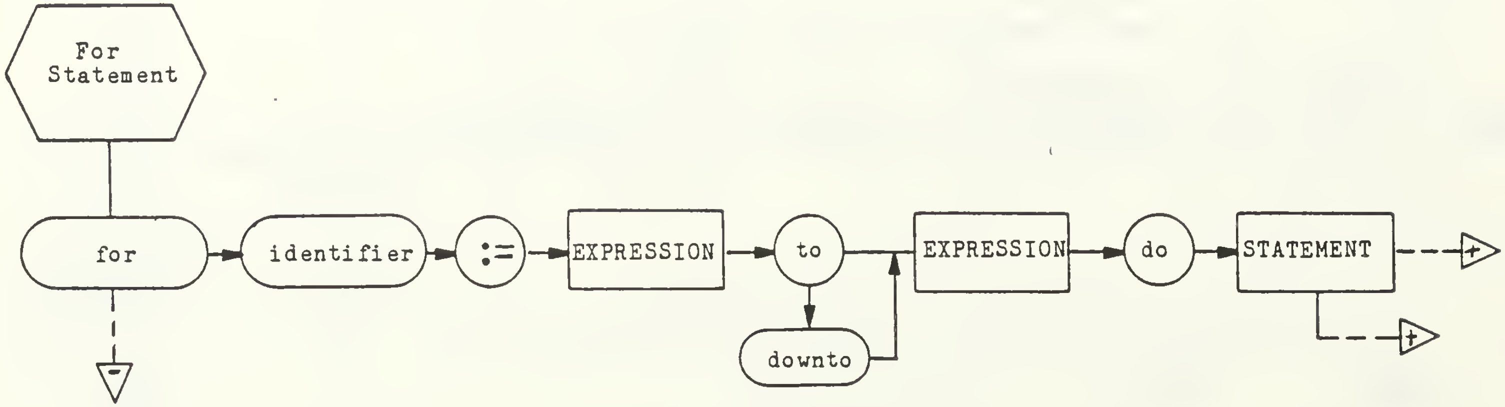

Term
FACTOR
*
/
FACTOR
div
mod
and
−
+

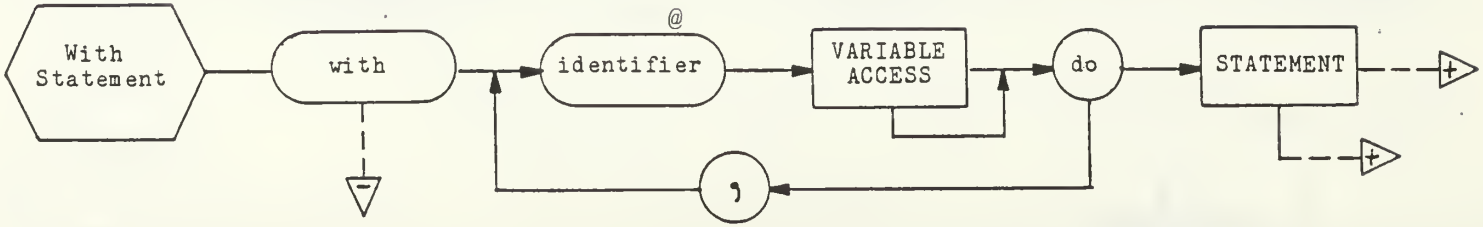

Constant
+ -
identifier
unsigned integer
unsigned real
@
identifier
@
unsigned integer
@
unsigned real
character string
+
-


Conformant Array Schema
packed
array
[
identifier
..
identifier
:
identifier
]
of
identifier
+
array
-
[
identifier
..
identifier
:
;
identifier
]
of
identifier
CONFORMANT ARRAY SCHEMA
+


If Statement
if
BOOLEAN EXPRESSION
then
STATEMENT
else
STATEMENT
+
+
+
-
Case Statement
case
EXPRESSION
of
CONSTANT
:
,
STATEMENT
;
@
CONSTANT
end
+
-
Repeat Statement
repeat
STATEMENT
until
BOOLEAN EXPERSSION
;
+
-


While Statement
while
BOOLEAN EXPRESSION
do
STATEMENT
+
+
-


For Statement
for
identifier
:=
EXPRESSION
to
downto
EXPRESSION
do
STATEMENT
+
+
-


With Statement
with
@
identifier
VARIABLE ACCESS
do
STATEMENT
,
+
+
-

APPENDIX C: PROGRAM LISTINGS

This program was coded using separate compilation on the UNIX operating system. Comments are provided for each procedure and function in the program to assist in understanding the purpose and design of each module. The program is divided into eight logical sections which appear in the following order:

(1) Main Routine and Declarations

(2) Lexical Routines

(3) Syntactic Routines

(4) Recovery Routines

(5) Error Processing Routines

(6) Output Routines

(7) Initializations

(8) Diagram Input File

MAIN ROUTINE

```
program syntacticanalyzer(input,output);

#include "global.h"
{ This is the main routine for the Syntactic Analyzer. The name of the
  file to be analyzed is read from the command line, along with any options
  which have been selected. Procedure parse is then called to perform
  the syntactic analysis. }

begin
  argv(1,filename);
  reset(input,filename);
  argv(2,option);
  printrecovset:= false;
  printhistory:= false;
  printbox:= false;
  printstack:= false;
  printlisting:= false;
  printposit:= false;
  i:= 1;
  while i <= totaloptions do
    begin
      if option[i] = 'r' then
        printrecovset:= true
      else if option[i] = 'h' then
        printhistory:= true
      else if option[i] = 'b' then
        printbox:= true
      else if option[i] = 's' then
        printstack:= true
      else if option[i] = 'l' then
        printlisting:= true
      else if option[i] = 'p' then
        printposit:= true
      else;
      i:= i + 1
    end;
  parse
end.
```

GLOBAL DECLARATIONS

```
const
          totaloptions      =     8;
          fileidlength      =     15;


type
          string            =     packed array[1..fileidlength] of char;
          switches          =     (prstack,prhalt,prresume,preof,preop);


var
          option            :     string;
          filename          :     string;
          i                 :     integer;
          printrecovset     :     boolean;
          printbox          :     boolean;
          printhistory      :     boolean;
          printstack        :     boolean;
          printlisting      :     boolean;
          printposit        :     boolean;



procedure parse; external;
```

COMMON DEFINITIONS

```
{******************** CONSTANTS AND TYPE DEFINITIONS ****************}
 {*******************************************************************}


    const
        reswordtotal  =  37;
        inpsymtotal   =  24;
        indextotal   =  24;
        statetotal   =  34;
        maxline      =  80;
        maxidlen     =   8;
        lexmsglength  =  50;
        maxname      =  25;
        namelength   =  31;
        totallexemes  =  70;

        intnil       =   0;
        exittrue     =  -1;
        exitfalse    =  -2;
        exiterror    =  -3;
        exitrecovery  =  -4;
        maxboxes     = 350;

       lineprintwidth = 132;
          lineoffset =  10;
      maxhistoryitems =   6;
           displayedge =  90;
              justify1 = 103;
              justify2 = 114;
              justify3 =  93;
                space1 =   3;
                space2 =   6;

    type
```

```
{*********************** LEXICAL DEFINITIONS ***********************}

    syntaxunit   =                    { lexemes }

                (doo,iff,inn,off,orr,too,andd,divv,endd,forr,modd,nill,
                 nott,sett,varr,casee,elsee,filee,gotoo,thenn,typee,withh,
                 arrayy,beginn,constt,labell,untill,whilee,writee,downtoo,
                 packedd,recordd,repeatt,forwardd,programm,writelnn,
                 ffunction,pprocedure,identifier,realconst,intconst,
                 stringconst,addop,mulop,relop,equals,colon,becomes,comma,
                 semicolon,period,range,pointer,lftparen,rtparen,
                 lftbracket,rtbracket,stop,endoffile,endmarker,illegal,
                 badcomment,badexpon,baddecpt,badsign,badstring,
                 zerostring,badexpart,baddecimal,nodigits);


    charset      = set of char;
    word         = packed array[1..maxidlen] of char;
    lexname       = packed array[1..maxname] of char;
    reswords      = packed array[0..reswordtotal] of word;
    lexvalue     = 0..totallexemes;
    lexconvert    = packed record
                        id: lexname;
                        su: syntaxunit;
                end;
    lexemelist    = packed array[1..namelength] of lexconvert;
    chindex       = packed record
                        ch : char;
                        val: integer;
                end;
    idlengths     = packed array[0..maxidlen] of integer;
    tableindex    = packed array[0..indextotal] of chindex;
    bufftype      = packed array[1..maxline] of char;
    lextable     = packed array[0..statetotal,0..inpsymtotal] of integer;
    lexmessage     = packed array[1..lexmsglength] of char;
    lexparams      = packed record
                        id      : reswords;
                        idlen    : idlengths;
                        tab      : lextable;
                        chrs     : tableindex;
                        list1    : lexemelist;
                        list2    : lexemelist;
```

```
                         eol     : boolean;
                         list    : boolean;
                         limit   : boolean;
                         badtext : boolean;
                         continue: boolean;
                         comments: boolean;
                         chpos   : integer;
                         chstart : integer;
                         lastpos : integer;
                         textend : integer;
                         letter  : charset;
                         number  : charset;
                         expon   : charset;
                         sign    : charset;
                         linebuf : bufftype;
                         auxbuf  : bufftype;
                         count   : integer;
                         linenum : integer;
                         oldline : boolean;
                         lasttok : lexvalue;
                         lastch  : char;
                         ch      : char;
                  end;

{********************* SYNTACTIC DEFINITIONS **********************}


       boxptr       = -4..maxboxes;
       boxtype      = (header,lexeme,nonterminal);
       boxname      = packed array[1..maxname] of char;
       box          = record
                      typ     : boxtype;
                      name    : boxname;
                      lexcode : integer;
                      nextptr : boxptr;
                      trueptr : boxptr;
                      falseptr: boxptr;
                   end;

       syntaxchart   = packed array[1..maxboxes] of box;
       headptr       = ^headlist;
       headlist      = packed record
                         name  : boxname;
```

```
                     boxnum: boxptr;
                     next   : headptr;
             end;
legalptr      = ^legallist;
legallist     = packed record
                     boxnum: boxptr;
                     next   : legalptr;
             end;
historyptr    = ^historyelement;
historyelement= record
                 name: boxname;
                 typ : boxtype;
                 next: historyptr;
             end;
stacktype     = (activation,recovery);
stackptr      = ^stackelement;
recovptr      = ^recovelement;
stackelement  = record
                  kind          : stacktype;
                  name           : boxname;
                  diagramhead : boxptr;
                  next          : stackptr;
                  returnaddr  : boxptr;
                  lasttrue     : boxptr;
                  histptr      : historyptr;
                  recovset     : recovptr;
                  currentrec  : stackptr;
             end;
namelist      = packed array[0..totallexemes] of boxname;
restartptr    = ^restartlist;
syntaxdata    = packed record
                     name  : namelist;
                     rstart: restartptr;
                     head  : headptr;
                     legal : legalptr;
                     total : integer;
                     last  : boxptr;
                     eop   : boxptr;
             end;
```

```
{********************** RECOVERY DEFINITIONS **********************}


        recovelement  = record
                          name      : boxname;
                          code      : integer;
                          diagrampos: boxptr;
                          parentrec : stackptr;
                          next      : recovptr;
                      end;
        restartlist   = packed record
                            token : lexvalue;
                            boxnum: boxptr;
                            next  : restartptr;
                      end;
        recoverposits = packed array[0..maxboxes] of boolean;
        usedsymbols   = packed array[0..totallexemes] of boolean;
        treeptr       = ^recovnode;
        recovnode     = packed record
                            code : integer;
                            true : treeptr;
                            false: treeptr;
                      end;
        recovset      = ^recovsymbols;
        recovsymbols  = packed record
                            symb: integer;
                            next: recovset;
                      end;
        recovdata     = packed record
                            points : recoverposits;
                            symbols: recovset;
                            used   : usedsymbols;
                      end;


{************************ ERROR DEFINITIONS *********************}


        garbledptr    = ^garbledtext;
        garbledtext   = packed record
                           junkstart: integer;
                           junkstop : integer;
                           symb     : lexvalue;
                           next     : garbledptr;
```

```
                end;
lexerrdata     = packed record
                     errpos : integer;
                     typ    : integer;
                     message: lexmessage;
                end;
lexerrorptr    = ^lexerrlist;
lexerrlist    = packed record
                     listing: lexerrdata;
                     next   : lexerrorptr;
                end;
errdata       = packed record
                     errstart : integer;
                     diagname : boxname;
                     starthist: historyptr;
                     endhist  : historyptr;
                     expected : legalptr;
                end;
errorptr      = ^errlist;
errlist      = packed record
                     listing: errdata;
                     next   : errorptr;
                end;
errormark      = ^sourceposit;
sourceposit    = packed record
                      pos : integer;
                      typ : char;
                      next: errormark;
                end;
errordata      = packed record
                     errptr     : errorptr;
                     lexerrptr  : lexerrorptr;
                     garbledlist: garbledptr;
                end;
```

```
{*********************** EXTERNAL DECLARATIONS **********************}

procedure  initialize     (var diagrams: syntaxchart; var lexx: lexparams;
                           var syntax: syntaxdata; var error: errordata;
                           var recov: recovdata);  external;

function   gettoken       (var lexx: lexparams; var error: errordata;
                           var diagrams: syntaxchart):lexvalue; external;

function   getchr         (var lexx: lexparams; var error: errordata;
                           var diagrams: syntaxchart): char; external;

procedure  lexicalerror   (var lexx: lexparams; num: lexvalue;
                           var error: errordata); external;

procedure  push           (typ: stacktype; var stack: stackptr;
                           name: boxname; pos,head: boxptr); external;

function   pop            (var stack: stackptr): boxptr; external;

procedure  update         (var stack: stackptr; loc: boxptr;
                           item: boxname; typ: boxtype); external;

procedure  insertlegal    (pos: boxptr; var p: legalptr); external;

function   findlegal      (pos: boxptr; p: legalptr;
                           var diagrams: syntaxchart): boolean; external;

function   getheadptr     (head: headptr; name: boxname): boxptr; external;

procedure  recover        (var stack: stackptr; var diagrams:syntaxchart;
                           var resumeptr: boxptr; var token: lexvalue;
                           var lexx: lexparams; var syntax: syntaxdata;
                           var error: errordata; var recov: recovdata); external;

procedure  errormessage   (var lexx: lexparams; var error: errordata;
                           var diagrams: syntaxchart); external;

procedure  recorderror    (var error: errordata; var lex: lexparams;
                           var stack: stackptr; var syntax: syntaxdata); external;

procedure  updatesource   (var error: errordata; badstuff: boolean;
                                       oldpos: integer; token: lexvalue;
                                       var lex: lexparams); external;

procedure     printmark               (errmarker: errormark); external;

function      findtextend             (var lexx: lexparams): integer; external;

procedure     outputhistory           (p: historyptr; q: errorptr; r: errormark;
                                       lastmark: integer); external;

procedure     outputlegal             (p: legalptr; q: errorptr; r: errormark;
                                       lastmark: integer; var diagrams: syntaxchart); external;

procedure     printset                (p: recovptr); external;

procedure     printhist               (p: historyptr); external;

procedure     printsyntax             (var diagrams: syntaxchart;
                                       var syntax: syntaxdata); external;

procedure     print                   (switch: switches; p: stackptr;
                                       var lexx: lexparams; var syntax: syntaxdata;
                                       var token: lexvalue); external;

function      length                  (name: boxname): integer; external;

{*********************************************************************}
```

```
#include "global.h"
#include "common.h"
```

LEXICAL ANALYSIS

```
{******************************************************************}
                     TEXT PROCESSING ROUTINES
{******************************************************************}
  procedure endline(var lexx: lexparams; var error: errordata;
                var diagrams: syntaxchart);
   { This module is called by getchr upon the first character read after
    processing has concluded on the current line.  If the "printlisting"
    command line switch has been set, then the buffered line of text is
    written and any accumulated text in the auxillary buffer is moved into
    the line buffer. The auxillary buffer holds the text which is read from
    the input file after eoln is true, providing temporary storage until all
    processing activities on the previous line (such as error messages) have
    been completed, i.e. it may not be until several characters into the
    succeeding line that an error is recognized on the current line.  The delay
    in handling end of line is accomplished via the lexical boolean variable
    "list".  The variable "oldline" used here is for the purpose of overriding
    the incremental line numbering in the event endline has been called due to
    reaching the 80 column boundary (maxline).  The variable "limit" indicates
    that maxline has been reached, but eol is not true.  The final action in
    this module is to call the error handler if any errors have been recorded. }

  const
     numberfield = 8;
  var
     i: integer;
  begin
    with lexx do
      begin
        if not oldline then
          begin
            linenum:= linenum + 1;
            if printlisting then
              write(linenum: numberfield,'  ')
            else
          end
        else
          write(' ': lineoffset);
      textend:= findtextend(lexx);
      for i:= 1 to lastpos-1 do
        begin
          if printlisting then
            write(linebuf[i])
          else;
          linebuf[i]:= ' ';
        end;
      if limit then
        begin
          if printlisting then
            writeln(linebuf[maxline])
          else;
          oldline:= true
        end
      else begin
             if printlisting then
               writeln(linebuf[lastpos])
             else;
             oldline:= false
           end;
      linebuf[maxline]:= ' ';
      for i:= 1 to maxline do
        begin
          linebuf[i]:= auxbuf[i];
          auxbuf[i]:= ' ';
        end;
      list:= false;
    end;
    with error do
      if (garbledlist <> nil) or (errptr <> nil)
          or (lexerrptr <> nil) then
        errormessage(lexx,error,diagrams)
      else
end;
```

```
{****************************************************************}
  function getchr;
  { This routine reads one character from the input file and returns it
    to the calling lexical analyzer subroutine. If the character position
    is at column 80 (maxline) or if eol is true, then the boolean "list" is
    set to signal that next time around the "endline" processing routine
    must be called. (Note: eol is set by the lexical analyzer when
    eoln(input) is true, but it is the next read operation, which will be the
    actual end of line position, when eol is recognized in getchr). A blank
    is the processing representation for both the eoln and eof characters.
    The character position counter (chpos) is reset to zero at end of line,
    and the lexical boolean variables which keep track of discarded text
    are set to enable the continuation of underlining, if currently enabled. }

  const
       tabadjust = 7;
       tabch     = 9;
  begin
    with lexx do
      begin
        if list then
          endline(lexx,error,diagrams);
        lastch:= ch;
        if (chpos = maxline) or (eol) then
          begin
            if not ((chpos = maxline) and (not eol)) then
              begin
                read(ch);
                limit:= false
              end
            else
              limit:= true;
            lastpos:= chpos;
            chpos:= 0;
            if comments or limit then
              begin
                chstart:= 1;
                if badtext then
                  continue:= true;
              end;
            list:= true;
          end
        else;
        if not eof(input) then
          if not eoln(input) then
            begin
              read(ch);
              chpos:= chpos + 1;
              if ord(ch) = tabch then
                chpos:= chpos + tabadjust;
              if not list then
                if ord(ch) <> tabch then
                  linebuf[chpos]:= ch
                else
              else
                 if ord(ch) <> tabch then
                   auxbuf[chpos]:= ch
                 else;
            end
          else
            ch:= ' '
        else
          ch:= ' ';
        getchr:= ch
      end;
  end;
{*****************************************************************}
{*****************************************************************}
                  LEXICAL ANALYZER UTILITIES
{*****************************************************************}
  procedure checkcaps(len: integer; var name: word);
  { This routine converts all characters to lower case, permitting recognition
    of reserved words which are capitalized or partially capitalized. Lower
    case symbols are used exclusively throughout the program. }

  const
       lowcase = 97;
       ascii   = 32;
  var
       i: integer;
  begin
    for i:= 1 to len do
      if ord(name[i]) < lowcase then
        name[i]:= chr(ord(name[i])+ascii)
      else
  end;
```

```
{*******************************************************************}
  function searchword(len: integer; ident: word; var lexx: lexparams): lexvalue;
  { This routine searches an array of reserved words, which are stored in
    increasing order of length, beginning with the first word in the list
    whose length is equal to the call parameter (thus only length "len" words
    are checked. If a word is found which matches "ident", then the array index
    is returned as the lexeme; otherwise, the identifier lexeme is returned. }

  var
       found: boolean;
           i: integer;
  begin
    with lexx do
      begin
        checkcaps(len,ident);
        i:= idlen[len-1];
        found:= false;
        while (not found) and (i < idlen[len]) do
          if ident = id[i] then
            found:= true
          else
            i:= i+1;
        if found then
          searchword:= i
        else
          searchword:=ord(identifier)
      end;
  end;
{*******************************************************************}
```

```
  function convert(c: char; var lexx: lexparams): integer;
  { This function is utilized by performscan to map input characters to
    integers in order to provide the vertical index into the lexical table.
    Columns include one for letters, one for numbers, one for illegal
    characters, and others as required to index each Pascal character. }

  const
      lettcolumn   = 22;
      numbcolumn   = 21;
      illegalch    = 23;
      indextotal   = 24;
  var
      i: integer;
  begin
    with lexx do
      begin
        i:= 0;
        while (chrs[i].ch <> c) and (i <= indextotal) do
          i:= i + 1;
        if i <= indextotal then
          convert:=chrs[i].val
        else if c in letter then
          convert:=lettcolumn
        else if c in number then
          convert:=numbcolumn
        else
          convert:=illegalch
      end;
    end;
{***************************************************************************}
  procedure checkcomment(c: char; var next: integer; var lexx: lexparams);
  { This procedure provides the capability to handle nested levels of comments
    by incrementing and decrementing a counter if the next state marks the
    beginning or end of a comment construct.  This feature comes in handy for
    commenting out sections of code that contain embedded comments. Both the
    primary and alternate comment symbols are checked here. }

  const
      comment = 15;
  begin
    with lexx do
      begin
        if c <> ' ' then
```

```
            if (c = '{') or ((c = '*') and (lastch =='(')) then
              count:= count + 1
            else if (c in [')','}']) and (next = 0) then
                   begin
                     count:= count - 1;
                     if count <> 0 then
                       begin
                         next:= comment;
                         comments:= true
                       end
                     else
                   end
                 else
          else;
        end
      end;
{********************************************************************}
    function adjustsymbol(var lexx: lexparams; symbol: lexvalue;
                          var error: errordata): lexvalue;
    { This function is the means by which lexical errors are suppressed.  If an
      error occurs in the lexical stage, it is recorded and entered into the
      lexical error linked list.  This routine then receives the erroneous
      lexeme and returns a syntactically valid lexeme to permit parsing to
      continue.  Also performed in this module is the conversion of the
      symbol "endmarker" into a representation for a "." .  This is necessary
      because a period which ends a program (i.e. "end.") needs to be treated
      differently than a period which is part of a field id.  Thus if the
      last lexeme was an "end", the assumption is that this symbol is a program
      end symbol, and the adjustment is made to return a lexeme for
      "endmarker" (the special period). }

    begin
      with lexx do
        if symbol = ord(period) then
          if lasttok = ord(endd) then
            adjustsymbol:=ord(endmarker)
          else
            adjustsymbol:=ord(period)
        else begin
               lexicalerror(lexx,symbol,error);
               if (symbol = ord(badexpon)) or (symbol = ord(baddecpt)) or
                  (symbol = ord(badsign))  or (symbol = ord(badexpart)) or
                  (symbol = ord(baddecimal)) or (symbol = ord(nodigits)) then
```

```
                   adjustsymbol:= ord(realconst)
              else if (symbol = ord(badstring)) or
                      (symbol = ord(zerostring)) then
                     adjustsymbol:= ord(stringconst)
              else
                adjustsymbol:= symbol;
            end;
    end;
{********************************************************************}
```

```
{*****************************************************************}
                    LEXICAL ANALYZER SUBROUTINES
{*****************************************************************}
  function processword(var lexx: lexparams; var error: errordata;
                       var diagrams: syntaxchart): lexvalue;
  { "Processword" is one of the two primary routines which comprise the
    scanner process.  This function is called by the main lexical routine
    (gettoken) whenever the current input character is a letter, which will
    result in generating either a reserved word or identifier. Processword
    consumes input until a character other than a letter or number is
    encountered (recognizing only the first 8) and stores the word in a
    buffer called "ident".  The routine searchword is then called to search
    the stored list of reserved words, based upon the passed length of ident
    to permit more efficient searching. }

  var
          i : integer;
      ident : word;
  begin
    with lexx do
      begin
        for i:= 1 to maxname do
          ident[i]:= ' ';
        i:= 0;
        repeat
          eol:= eoln(input);
          if i< maxidlen then
            begin
              i:= i+1;
              ident[i]:= ch
            end;
          ch:= getchr(lexx,error,diagrams);
        until not ((ch in letter) or (ch in number)) or (eol) or (eof(input));
        processword:= searchword(i,ident,lexx)
      end;
  end;
{*****************************************************************}
```

```
function performscan(var lexx: lexparams; var error: errordata;
                     var diagrams: syntaxchart): lexvalue;
{ This function is the second of the lexical analysis routines, generating
  lexical tokens for all language symbols except word symbols, including
  real, integer, and string constants. The heart of this routine is a two-
  dimensional table, indexed by input character and state number, which
  simulates the performance of an FSA on the standard Pascal character set.
  In addition to generating tokens, the table also provides the means for
  consuming source text which is contained within comment brackets. A repeat-
  until construct is utilized to effect the state to state movement thru the
  table. Transitions continue until a -1 sentinel (stopstate) is reached, at
  which point the rightmost column (tokencol) contains the lexeme for the
  symbol which has been recognized. Errors such as string quotes, missing
  comment close, and real constant errors are also represented by integer
  codes, but they are adjusted in the lexical stage and returned to the parser
  as valid lexemes.

  A note about end-of-line:  the variable "eol" is set to the value of eoln
  upon each entry into the table. This value, rather than eoln, is used for
  end of line determination, since once the last character has been read, eoln
  is false. }

const       tokencol = 24;
          ordrangech = 31;
      lookaheadstate = 31;
        realerrstate = 32;
            comment1 = 15;
            comment2 = 16;
           stopstate = -1;
var
    oldstate: integer;
    newstate: integer;
begin
  oldstate:= 0;
  newstate:= 0;
  with lexx do
    begin
      repeat
        eol:= eoln(input);
        oldstate:= newstate;
        newstate:= tab[oldstate,convert(ch,lexx)];
        if (newstate <> stopstate) or (oldstate >= realerrstate) then
          if newstate = lookaheadstate then
            begin
```

```
              oldstate:= newstate;
              newstate:= stopstate;
              if ch = ')' then
                ch:= ']'
              else
                ch:= chr(ordrangech)
            end
          else begin
                 if (newstate = comment1) or (newstate = comment2) or
                    (newstate = 0) then
                   begin
                     if newstate <> 0 then
                       comments:= true
                     else
                       comments:= false;
                     chstart:= chstart+1;
                     checkcomment(ch,newstate,lexx)
                   end
                 else if (lastch in expon) and (ch in sign) then
                        if oldstate >= realerrstate then
                          newstate:= oldstate
                        else
                      else;
                 if newstate <> stopstate then
                   ch:= getchr(lexx,error,diagrams)
                 else;
                 if comments then
                   eol:= false
                 else
               end
        else
      until (newstate = stopstate) or eol or eof(input);
      if (eol) and (newstate <> stopstate) then
        performscan:= tab[newstate,tokencol]
      else
        performscan:= tab[oldstate,tokencol];
    end;
end;
```

LEXICAL ANALYZER DRIVER

```
{**************************************************************************}
  function gettoken;
  { This is the controlling routine for the lexical stage. The appropriate
    subroutine (processword for a letter, performscan for all others) is called
    for character-by-character scanning of the source text. The returned token
    is then forwarded to the parser for use in the syntactic analysis. In the
    event that no token is returned (blank line, etc), a recursive call to
    gettoken is executed. Upon reaching end of file, an end-of-file token is
    sent to the parser. }

  var
      symbol: lexvalue;
  begin
    with lexx do
      begin
        if not eof(input) then
          begin
            chstart:= chpos;
            if ch in letter then
              symbol:= processword(lexx,error,diagrams)
            else
              symbol:= performscan(lexx,error,diagrams);
            if symbol = ord(stop) then
              symbol:= gettoken(lexx,error,diagrams);
            if (symbol > ord(endoffile)) or (symbol = ord(period)) then
              symbol:= adjustsymbol(lexx,symbol,error)
            else;
            lasttok:= symbol;
          end
        else begin
               symbol:= ord(endoffile);
               if lasttok = ord(endoffile) then
                 endline(lexx,error,diagrams);
               lasttok:= symbol
             end;
      end;
    gettoken:= symbol;
  end;
{*************************************************************************************}
```

```
#include "global.h"
#include "common.h"
```

SYNTACTIC ANALYZER

```
{*******************************************************************}
                    STACK MANIPULATION ROUTINES
{*******************************************************************}
 procedure push;
 { This routine is called by both the parsing and recovery modules to
  push a diagram activation record onto the stack.  Two types of records
  may be pushed:  activation or recovery.  If the record is to be pushed
  for normal execution (type activation), then all fields except the
  "recovset" and "currentrec" are applicable.  If the record is a recovery
  type, then the "recovset" pointer is used to point to the set of
  recovery symbols, and the "currentrec" field points to that level of
  stack to which the symbol belongs.  The constant "intnil" represents a
  null initialization for integer pointers in order to distinguish them
  from the dynamic pointer, "nil". }

 var
     p: stackptr;
 begin
  new(p);
  p^.kind:= typ;
  p^.name:= name;
  p^.returnaddr:= pos;
  p^.diagramhead:= head;
  p^.next:= stack;
  p^.lasttrue:= intnil;
  p^.histptr:= nil;
  p^.recovset:= nil;
  p^.currentrec:= stack;
  stack:= p
 end; {push}
{*******************************************************************}
 function pop;
 { This routine returns an integer pointer which represents the return
  address for the level of stack activation which has just been
  completed, i.e. this pointer determines the position in the transition
  diagrams from which the parse will resume.  If the stack is empty,
  this is conveyed to the parser by returning "intnil". }
  var
    p: stackptr;
  begin
   p:= stack;
   stack:= stack^.next;
   if stack <> nil then
    pop:= p^.returnaddr
   else
    pop:= intnil;
   dispose(p)
  end;
{*********************************************************************}
  procedure update;
  { This routine is responsible for updating the "history list".  The
   history pointer (variable "histptr") points to a linked list
   which contains one node for each box which has been successfully
   traversed while the corresponding activation record has been on
   the stack.  This information is later used by the error handler
   to build any error message which may be required in connection with
   the current stack activation.  The term "junk" is inserted into the
   list if the history of the activation contains a segment where source
   text was discarded by the recovery process. }

  var
    p,q: historyptr;
  begin
   if stack <> nil then
    with stack^ do
     begin
      if loc <> intnil then
       lasttrue:= loc;
      if loc <> 1 then
       begin
        new(p);
        p^.name:= item;
        p^.typ:= typ;
        p^.next:= nil;
        if histptr = nil then
         histptr:= p
        else begin
           q:= histptr;
           while q^.next <> nil do
            q:= q^.next;
```

```
                      if (q^.name = 'junk') and
                         (p^.name = 'junk') then
                        q:= p
                      else
                        q^.next:= p
                    end
             end
           else;
           if printhistory then
             printhist(histptr)
           else
         end
     else
   end; { update }
{*********************************************************************}
```

```
{*******************************************************************}
                              PARSER
{*******************************************************************}
  procedure parse;
  { This is the parsing mechanism for the Syntactic Analyzer.  Traversal
    through the transition diagrams is contolled iteratively by a repeat-until
    loop, and is terminated when the parsing stack has been emptied.  On each
    pass through the loop, one of three box types may be encountered: header,
    nonterminal, or lexeme.  If it is a header, the location pointer is set
    to the first box in the diagram;  if it is an nonterminal, then an
    activation record is pushed onto the stack, and the location pointer
    is set to the header box of the new diagram to be traversed;  if it is
    a lexeme, then the location pointer is set to either the box's true or
    false exit pointer, depending upon whether the currently held lexeme
    matches that associated to the box.  If a true exit is taken, an update
    routine is called to record the true exit in the history list.  If the
    exit is false and the box is a lexeme, then the set of all possible
    legal symbols (held in the variable "syntax.legal") is updated in the
    "legal" list which contains the symbols which "could have been".  Calls
    to various print utilities (if desired for debugging) are also performed
    from this module in response to command line switch settings. }

  var
      lexx      : lexparams;
      diagrams : syntaxchart;
      p         : boxptr;
      location : boxptr;
      returnptr: boxptr;
      token     : lexvalue;
      stack     : stackptr;
      errors    : errordata;
      syntax    : syntaxdata;
      recov     : recovdata;
  begin
    initialize(diagrams,lexx,syntax,errors,recov);

    { Initialize the parsing stack, push the "Program" transition
      diagram activation record onto the stack, and call lexx for
      the first lexeme.  The initial call to update is required to
      provide the recovery routine with a non-zero last true exit
      in the case where recovery mode may be entered immediately,
      i.e. missing "program". }
```

```
stack:= nil;
p:= 1;
push(activation,stack,diagrams[p].name,p,p);
update(stack,p,diagrams[p].name,diagrams[p].typ);
token:= gettoken(lexx,errors,diagrams);

{ Begin syntactic analysis by following the location pointer
  through the transition diagrams, which are accessed via the
  variable "diagrams". }

repeat
  with diagrams[p] do
    begin
      if typ = header  then
        location:= nextptr
      else if typ = nonterminal then
             begin
               push(activation,stack,name,p,nextptr);
               location:= nextptr
             end
      else if token = lexcode then
             begin
               location:= trueptr;
               update(stack,p,name,typ);
               token:= gettoken(lexx,errors,diagrams);
               syntax.legal:= nil;
             end
      else begin
             location:= falseptr;
             if not(findlegal(p,syntax.legal,diagrams)) then
               insertlegal(p,syntax.legal)
             else
           end;

      repeat
        if (location = exittrue) or (location = exitfalse) then
          repeat
            returnptr:=pop(stack);
            if returnptr <> exitrecovery then
              if returnptr <> intnil then
                if location = exittrue then
                  begin
                    location:=diagrams[returnptr].trueptr;
                    update(stack,returnptr,diagrams[returnptr].name,
                                        diagrams[returnptr].typ);
             end
            else
             location:=diagrams[returnptr].falseptr
           else
            location:= intnil
          else
           location:= exitrecovery
        until ((location <> exittrue) and (location <> exitfalse)) or
              (location = exitrecovery)
       else;

       { Check to see if either an error has been detected or if
         parsing which was previously initiated by a restart symbol
         has been completed, in which case control is shifted back to
         the recovery mode by encountering an "exitrecovery". }

       if (location = exiterror) or (location = exitrecovery) then
         begin
           if printposit then
             print(prhalt,stack,lexx,syntax,token);
           if printstack then
             print(prstack,stack,lexx,syntax,token);
           recover(stack,diagrams,location,token,lexx,syntax,errors,recov);
           if location <> intnil then
             if printstack then
               print(prstack,stack,lexx,syntax,token)
             else
           else
         end
       else
      until (location <> exittrue) and (location <> exitfalse);

      { Go to the next diagram box as determined by the location
        pointer.  Parsing terminates if the stack is empty. }

      p:= location;
    end;
  until (stack = nil)
 end;
{*******************************************************************}
```

```
#include "global.h"
#include "common.h"
```

ERROR RECOVERY ROUTINES

```
{**********************************************************************}
                      ERROR RECOVERY UTILITIES
{**********************************************************************}
   function makenode(boxnum: boxptr): treeptr;
   { This function creates a node of the recovery set tree, which is
     formed by the "buildset" and "genrecovset" routines.  This tree is
     constructed dynamically and represents a traversal of the syntax
     transition diagrams in collecting the set of recovery symbols.  Each
     node in the tree has two sons, one each for the true and false box
     exit paths. }

   var
      p: treeptr;
   begin
     new(p);
     p^.code:= boxnum;
     p^.true:= nil;
     p^.false:= nil;
     makenode:= p
   end;
{**********************************************************************}
   procedure addsymbol(rp: stackptr; var diagrams: syntaxchart;
                   loc: boxptr);
   { This procedure adds a recovery symbol to the resynchronization
     set, which is represented by a linked list and is pointed to by the
     recovery set pointer of the current recovery activation.  Symbol
     information includes the name, parent diagram, position within that
     diagram, and lexeme code. }

   var  p,q: recovptr;
   begin
     new(p);
     p^.name:= diagrams[loc].name;
     p^.code:= diagrams[loc].lexcode;
     p^.diagrampos:= loc;
     p^.parentrec:= rp^.currentrec;
     p^.next:= nil;
     if rp^.recovset = nil then
```

```
      rp^.recovset:= p
    else begin
          q:= rp^.recovset;
          while q^.next <> nil do
            q:= q^.next;
          q^.next:= p
        end
  end;
{*****************************************************************}
  function searchlist(var rp: stackptr; token: lexvalue): boxptr;
  { This function searches the recovery symbol set, once for each lexeme
    consumed during the recovery process.  If the currently held lexeme
    matches one of the recovery symbols, the recovery stack pointer is
    set to the level of stack pointed to by the symbol's "parent record"
    pointer, and the transition diagram position for this symbol (which is
    where parsing will resume) is returned to the calling routine.  If
    no symbol is found, the "intnil" pointer is returned.  }

  var
      found: boolean;
          p: recovptr;
  begin
    p:= rp^.recovset;
    found:= false;
    while (p <> nil) and (not found) do
      if p^.code = token then
        found:= true
      else
        p:= p^.next;
    if p = nil then
      searchlist:= intnil
    else begin
          rp:= p^.parentrec;
          searchlist:= p^.diagrampos;
        end;
  end;
```

```
{****************************************************************}
  function computepos(var diagrams: syntaxchart; newpos: boxptr;
                    token: lexvalue): boxptr;
  { This routine is used to compute the proper resumption point in the
    transition diagrams if a restart symbol was found.  If the symbol is
    not the first box in the diagram, then the false exit path through the
    diagram is followed until the symbol is found. }

  var
     pos: integer;
  begin
    if diagrams[newpos+1].lexcode = token then
      computepos:= newpos + 1
    else begin
          pos:= newpos+1;
          repeat
            pos:= diagrams[pos].falseptr
          until (diagrams[pos].lexcode = token);
          computepos:= pos
        end
  end;
{****************************************************************}
  function getheadptr;
  { This routine returns the starting position of a diagram header box.  This
    function is called by the recovery module to determine a parsing resumption
    point following a restart recovery which requires modifying the stack by
    pushing a new activation record.  Since a separate nonterminal for "Boolean
    expression" is not used (i.e. there is no diagram), a check is made here to
    return the expression header address in that situation. }

  var
      found: boolean;
          p: headptr;
  begin
    p:= head;
    found:= false;
    while not found do
      if p^.name = name then
        found:= true
      else if (p^.name = 'expression') and
              (name = 'Boolean_expression') then
        found:= true
      else
      p:= p^.next;
    getheadptr:= p^.boxnum
  end;
{*********************************************************************}
  function searchrestart(head: restartptr; code: lexvalue): boxptr;
  { This routine is called by the recovery module to see if the currently held
    lexeme is a member of the restart symbol set. }

  var
      found: boolean;
          p: restartptr;
  begin
    p:= head;
    found:= false;
    while (not found) and (p <> nil) do
      if p^.token = code then
        found:= true
      else
        p:= p^.next;
    if found then
      searchrestart:= p^.boxnum
    else
      searchrestart:= intnil
  end;
{*********************************************************************}
  function checkrecov(head: recovset; code: integer): boolean;
  { This routine is called by the recovery module to see if the currently held
    lexeme is a member of the resynchronization symbol set. }

  var
      found: boolean;
          p: recovset;
  begin
    p:= head;
    found:= false;
    while (not found) and (p <> nil) do
      if p^.symb = code then
        found:= true
      else
        p:= p^.next;
    checkrecov:= found;
  end;
```

```
{*********************************************************************}
{*********************************************************************}
                    ERROR RECOVERY SUBROUTINES
{*********************************************************************}
  procedure buildset(p: treeptr; newbox: boxptr; branch: char; var diagrams:
                syntaxchart; var stack,rp: stackptr; var recov: recovdata);
  { This routine is called by "genpreorder" to construct a "tree" data
    structure which is used to generate the error recovery set.  The tree
    is built by making a node for each box in the transition diagram which
    is positioned along either a true or false exit path from the point where
    the last true exit was taken.  If the box corresponds to a resynchronization
    symbol, then the "addsymbol" routine is called to update the recovery set.
    The boolean recovery point and used symbol arrays are then updated
    accordingly.  The tree construction is terminated when all boxes within
    the diagram in the forward direction from the error position have been
    examined. }

  var
     newsymbol: treeptr;
  begin
    if (newbox > 0) and
       ((newbox <> diagrams[stack^.lasttrue].falseptr) or (branch = 't')) then
      if not recov.points[newbox] then
        begin
          if branch = 't' then
            begin
              p^.true:= makenode(newbox);
              newsymbol:= p^.true
            end
          else begin
                 p^.false:= makenode(newbox);
                 newsymbol:= p^.false
               end;
          if diagrams[newsymbol^.code].typ = lexeme then
            if checkrecov(recov.symbols,diagrams[newsymbol^.code].lexcode) then
              if not (recov.used[diagrams[newsymbol^.code].lexcode]) then
                begin
                  addsymbol(rp,diagrams,newbox);
                  recov.used[diagrams[newsymbol^.code].lexcode]:= true
                end
              else
            else
          else;
      recov.points[newsymbol^.code]:= true
     end
    else
   else
  end;
{********************************************************************}
  procedure genpreorder(p: treeptr; var stack: stackptr; var rp: stackptr;
                var diagrams:syntaxchart; var recov: recovdata);
  { This routine controls the recovery symbol generation process by
   creating and traversing a tree data structure in preorder.  This
   recursive procedure follows the standard "root-left-right" preorder
   scheme where left, in this case, represents a true exit path and right
   represents a false exit path. }

  begin
   if p <> nil then
    with diagrams[p^.code] do
     begin
      buildset(p,trueptr,'t',diagrams,stack,rp,recov);
      genpreorder(p^.true,stack,rp,diagrams,recov);
      buildset(p,falseptr,'f',diagrams,stack,rp,recov);
      genpreorder(p^.false,stack,rp,diagrams,recov);
     end
  end;
{********************************************************************}
```

```
procedure genrecovset(var stack: stackptr; var diagrams: syntaxchart;
                                            var recov: recovdata);
{ This is the driver for the recovery symbol generation process.  The
  purpose of this procedure is to "walk" down the parsing stack (whose
  top at time of call is the most recent recovery activation record) and
  generate any potential recovery symbols for each activation level.  This
  walk down the stack concludes when either the last activation level has
  been reached or a recovery record from a previous recovery is encountered.
  The final step of this routine joins this newly derived set with any
  existing set which may already be present, i.e. the recovery set pointer
  is adjusted, if necessary to "hook" onto the beginning of the existing
  set, thus forming a "union" of recovery symbols.  An important variable
  used here (and in some of the other recovery subroutines above) is "rp",
  or the recovery pointer, which provides the current point of reference
  (i.e. what is the current level of stack) so as to act as a "movable"
  pointer while the variable "stack" remains fixed at the top. }

var
    top: stackptr;
      i: integer;
      p: treeptr;
      q: recovptr;
     rp: stackptr;
begin

 { initialize the boolean recovery point and used symbol arrays to indicate
   that no diagram position has yet to be investigated as a possible recovery
   point, and check the first stack level }

  for i:= 0 to maxboxes do
    recov.points[i]:= false;
  for i:= 0 to totallexemes do
    recov.used[i]:= false;
  rp:= stack;
  stack:= stack^.next;
  p:= makenode(stack^.lasttrue);
  genpreorder(p,stack,rp,diagrams,recov);

 { now that the first level has been checked, start walking down }

  stack:= stack^.next;
  if stack <> nil then
    repeat
      top:= stack;
```

```
          if stack^.kind <> recovery then
            begin
              rp^.currentrec:= stack;
              if stack^.lasttrue <> intnil then
                begin
                  p:= makenode(stack^.lasttrue);
                  genpreorder(p,stack,rp,diagrams,recov);
                end
              else;
            end
          else begin

                { join the sets, if required }

                 q:= rp^.recovset;
                 if q <> nil then
                   begin
                     while q^.next <> nil do
                       q:= q^.next;
                     q^.next:= stack^.recovset;
                   end
                 else
               end;
          stack:= stack^.next
        until (stack = nil) or (top^.kind = recovery)
      else;
      stack:= rp;
      if printrecovset then
        printset(rp^.recovset);
    end;
{**********************************************************************}
```

```
function performrecovery(var stack: stackptr; var diagrams: syntaxchart;
                         var token: lexvalue; var error: errordata;
                         var syntax: syntaxdata; var lex: lexparams): boxptr;
{ This routine returns the position in the transition diagrams where normal
  parsing will resume. The following recovery decisions and actions are
  either initiated or performed here: 1) determine whether or not the current
  lexeme is a member of the "restart" set and if so, initiate action to
  get the appropriate activation record onto the stack, and compute the
  resumption point for parsing on this symbol, 2) initiate a search of the
  recovery set for a match with the current lexeme and if found, return its
  diagram postion, 3) interface with a display routine ("updatesource") which
  keeps track of the "bad text" as each token is discarded during the recovery
  for later underlining of the affected source. One variable used here whose
  use may not be easily understood is "oldpos", which is necessary to hold
  the starting position of each lexeme prior determining whether or not it will
  be thrown away and, therefore, underlined. Control within this module is
  accomplished via a repeat-until loop, meaning, consume lexemes in the input
  until one is found which meets the recovery criteria discussed above. }

var
    returnptr: boxptr;
           rp: stackptr;
       newpos: boxptr;
       oldpos: integer;
begin
  rp:= stack;
  oldpos:= 0;
  lex.badtext:= true;
  repeat
    updatesource(error,lex.badtext,oldpos,token,lex);
    returnptr:= searchlist(rp,token);
    stack:= rp;
    if returnptr = intnil then
      begin
        newpos:= searchrestart(syntax.rstart,token);
        if newpos <> intnil then
          begin
            push(activation,stack,diagrams[newpos].name,exitrecovery,newpos);
            returnptr:= computepos(diagrams,newpos,token);
          end
        else
      end
    else;
    if returnptr <> intnil then
```

```
         begin
           if returnptr <> syntax.last then
             if printposit then
               print(prresume,stack,lex,syntax,token)
             else
           else;
           lex.badtext:= false;
           updatesource(error,lex.badtext,oldpos,token,lex);
         end
       else begin
              lex.badtext:= true;
              update(rp^.next,intnil,'junk',lexeme)
            end;
       oldpos:= (lex.chpos-1)+lineoffset;
       token:= gettoken(lex,error,diagrams);
       syntax.legal:= nil;
     until (returnptr <> intnil) or (returnptr = syntax.last);
     if returnptr = syntax.last then
       print(preof,stack,lex,syntax,token)
     else;
     update(stack,returnptr,diagrams[returnptr].name,diagrams[returnptr].typ);
     performrecovery:= diagrams[returnptr].trueptr;
   end;
{********************************************************************}
```

```
{********************************************************************}
                      ERROR RECOVERY DRIVER
{********************************************************************}
  procedure recover;
  { This is the driver for the error recovery mechanism.  If recovery mode
    is being entered due to the occurence of a new error, then a recovery record
    is pushed onto the stack, all of the error data needed for producing an error
    message is computed and saved, the recovery set is generated, and the
    serach begins for a resynchronization symbol.  If recovery mode is being
    reentered, having just completed parsing a segment of text which began as a
    result of a previously found restart symbol, then the recovery resumes by
    searching the recovery set extending from the old record which has just
    reappeared at the top of the parsing stack.  The call to print in this module
    is for the purpose of informing the user that an "end of program" (end.) has
    been detected.  Processing continues, however, to detect any errors in the
    remaining text. }

  begin
    if stack^.kind <> recovery then
      with stack^ do
        begin
          if lasttrue = syntax.eop then
            print(preop,stack,lexx,syntax,token)
          else;
          push(recovery,stack,name,lasttrue,diagramhead);
          recorderror(error,lexx,stack,syntax);
          genrecovset(stack,diagrams,recov);
      end
    else;
    resumeptr:= performrecovery(stack,diagrams,token,error,syntax,lexx);
  end;
{********************************************************************}
```

```
#include "global.h"
#include "common.h"
```

ERROR HANDLING ROUTINES

```
{***********************************************************************}
                  ERROR MESSAGE PREPARATION ROUTINES
{***********************************************************************}
   These routines are concerned with performing linked list operations
   required for preparation of the error messages.  Some of these
   routines are utilized in connection with the "legal symbol list",
   which is used to produce the error narrative that lists those
   symbols which would have been syntactically legal at the point of
   error detection.  Additionally, the elements of the history list,
   which contains those syntactic units which have been successfully
   recognized prior to the point of error, are extracted and assigned
   to an error message pointer for later display.
{***********************************************************************}
  function findlegal;
  { This function searches the legal list and returns a boolean which is
    used to prevent insertion of duplicate box names. }

  var
      found: boolean;
  begin
    found:= false;
    while (p <> nil) and (not found) do
      if diagrams[p^.boxnum].name = diagrams[pos].name then
        found:= true
      else
        p:= p^.next;
    findlegal:= found;
  end;
{***********************************************************************}
  procedure insertlegal;
  { This procedure adds an element to the legal list and is called by
    both the parser and error handler.  The parser inserts a symbol into
    list upon exiting false from a lexeme box, and the error handler
    determines the remainder of the symbols by examining those which were
    not checked during normal execution. }

  var
      q,r: legalptr;
   begin
     new(q);
     q^.boxnum:= pos;
     q^.next:= nil;
     if p = nil then
       p:= q
     else begin
            r:= p;
            while r^.next <> nil do
               r:= r^.next;
            r^.next:= q
          end
   end;
{*******************************************************************}
   procedure recorderror;
   { This is the main routine for error message preparation.  The following
     actions are performed here:  1) the source position of the error is
     recorded, 2) the name of the diagram in which the error occured is
     saved (to output "bad..."), 3) the end of the history list is saved,
     4) the contents of the legal list are saved, and 5) all of the various
     components of the message are saved in a message record (the variable
     "listing" below), which is a member of an error list for the current
     line.  Access to the messages for the line is provided through the
     pointer variable "errptr".  }

   var
       p,q: errorptr;
         r: historyptr;
         s: legalptr;
   begin
     with lex,error,stack^.next^ do
       begin
         new(p);
         with p^.listing do
           begin
             if list then
               errstart:= lastpos+1
             else
               errstart:= chpos;
             diagname:= name;
             starthist:= histptr;
             r:= starthist;
             if r <> nil then
```

```
               begin
                 while r^.next <> nil do
                   r:= r^.next;
                 endhist:= r
               end
             else;
             s:= syntax.legal;
             expected:= nil;
             while s <> nil do
               begin
                 insertlegal(s^.boxnum,expected);
                 s:= s^.next
               end;
           end;
         p^.next:= nil;
         if errptr = nil then
           errptr:= p
         else begin
                q:= errptr;
                while q^.next <> nil do
                  q:= q^.next;
                q^.next:= p
              end;
       end;
   end;
{********************************************************************}
```

```
procedure lexicalerror;
{ This routine records lexical stage errors and enters them into a
  lexical error linked list.  This list is later merged with the syntactic
  error list permitting output routines to traverse one list in displaying
  all the error information occuring on a given line.  Based on the call
  parameter indicating lexical id, the appropriate message is retrieved
  and stored for output at end of line.  }

var
    p,q: lexerrorptr;
   text: lexmessage;

  procedure getmessage(num: lexvalue; var text: lexmessage);

  begin
    if num = ord(illegal) then
      text:= 'illegal character(s)'
    else if num = ord(badcomment) then
      text:= 'unclosed comment detected'
    else if num = ord(badexpon) then
      text:= 'digit,+,- must follow "e"'
    else if num = ord(baddecpt) then
      text:= 'digit(s) must follow dec pt.'
    else if num = ord(badsign) then
      text:= 'digit(s) must follow sign in exponent'
    else if num = ord(badstring) then
      text:= 'unclosed string quote at end of line'
    else if num = ord(zerostring) then
      text:= 'zero string constant not allowed'
    else if num = ord(badexpart) then
      text:= 'illegal exponent in real constant'
    else if num = ord(baddecimal) then
      text:= 'illegal rt side of decimal pt.'
    else if num = ord(nodigits) then
      text:= 'digit(s) must come before dec pt.'
  end; {get message}

begin {lexicalerror}
  with error,lexx do
    begin
      new(p);
      with p^.listing do
        begin
          if list then
        errpos:= lastpos+1
       else
        errpos:= chpos;
       typ:= num;
       getmessage(num,text);
       message:= text;
      end;
     p^.next:= nil;
     if lexerrptr = nil then
      lexerrptr:= p
     else begin
         q:= lexerrptr;
         while q^.next <> nil do
          q:= q^.next;
         if q^.listing.typ <> ord(badcomment) then
          q^.next:= p
         else
        end
    end;
  end; {lexicalerror}
{*******************************************************************}
  procedure collecterrors(q: lexerrorptr; r: errorptr; var s: errormark;
              var lastmark: integer);
  { This routine takes the input lexical and syntactic error pointers
   (locally as pointers "q" and "r" respectively) and merges the
   error position information from the two lists.  Lexical errors are
   noted with a 'l' and syntactic with an 's', in the event that multiple
   errors occur at the same point on the line (and if so, lexicals will
   be output first).  This information is later used by the error message
   driver routine to control the order of the message output processing.
   The variable "listing" used here, and in other error message routines,
   is the record of error information for each error, which contains the
   history list pointer, legal list pointer, diagram name, and the error
   position. }

  var
    p,t: errormark;
  begin
   while (q <> nil) or (r <> nil) do
   begin
    new(p);
    if (q <> nil) and (r <> nil) then
     if q^.listing.errpos <= r^.listing.errstart then
```

```
           begin
             p^.pos:= q^.listing.errpos;
             p^.typ:= 'l';
             q:= q^.next
           end
         else begin
                p^.pos:= r^.listing.errstart;
                p^.typ:= 's';
                r:= r^.next
              end
       else if (q <> nil) then
              begin
                p^.pos:= q^.listing.errpos;
                p^.typ:= 'l';
                q:= q^.next
              end
       else begin
              p^.pos:= r^.listing.errstart;
              p^.typ:= 's';
              r:= r^.next
            end;
       if s = nil then
         s:= p
       else begin
              t:= s;
              while t^.next <> nil do
                t:= t^.next;
              t^.next:= p
            end;
       if (q = nil) and (r = nil) then
         lastmark:= p^.pos + lineoffset-1
       else
     end;
   end;
{*******************************************************************}
```

```
{*********************************************************************}
                    ERROR MESSAGE DISPLAY UTILITIES
{*********************************************************************}
  procedure updatesource;
  { This routine records the line start and stop positions for those
    lexemes which are discarded during error recovery.  This information
    is later used by the "underline" routine in marking the affected text.
    The algorithm here is as follows: 1) if the call parameter badstuff
    is false (meaning recovery has occured), then find the last element in
    the "garbled" linked list and record the "junk" stop position; if this
    posit equals the start position, then recovery occured immediately
    without consuming text and the stop posit becomes one less than the start
    to indicate that no underlining should be performed; otherwise, mark the
    stop posit. 2) if the call is true, but no stop was enterd for the last
    item in the list, then a new list element is not neceasary since the
    recovery has not yet occured (thus underlining should continue).  3) and
    finally, if the call is true and the list is empty, create a new node and
    enter the start position. }

  var
    p,q: garbledptr;
  begin
   with error,lex do
    begin
     if badstuff then
      if garbledlist = nil then
       begin
        new(p);
        p^.next:= nil;
        p^.symb:= token;
        if continue then
         begin
          i:= 1;
          while linebuf[i] = ' ' do
           i:= i+1;
          p^.junkstart:= lineoffset + i;
          continue:= false
         end
        else
         p^.junkstart:= chstart + lineoffset;
        p^.junkstop:= 0;
        garbledlist:= p
       end
      else begin
          p:= garbledlist;
          while p^.next <> nil do
           p:= p^.next;
          if p^.junkstop <> 0 then
           begin
            new(q);
            q^.next:= nil;
            q^.symb:= token;
            q^.junkstart:= chstart+ lineoffset;
            q^.junkstop:= 0;
            p^.next:= q
           end
          else
         end
     else begin
         p:= garbledlist;
         while p^.next <> nil do
          p:= p^.next;
         if token = p^.symb then
          p^.junkstop:= p^.junkstart-1
         else
          p^.junkstop:= oldpos;
        end;
    end
  end;
{********************************************************************}
  procedure printmark;
  { The purpose of this routine is to display and align the vertical lines
   which extend downward from the text source line from each error position
   on the line.  The call parameter for this module is a pointer to a
   list of error positions on the source line.  A counter is set to the left
   edge of the display and a vertical bar is printed each time the counter
   position equals one of the stored error positions in the list. }

  var
    lastpos: integer;
         i: integer;
         p: errormark;
  begin
   lastpos:= 0;
   i:= 10;
   p:= errmarker;
```

```
     while p <> nil do
       begin
         if i = p^.pos + lineoffset-1 then
           begin
             if i <> lastpos then
               begin
                 if (lastpos = 0) and (i = lineoffset) then
                   write('|':lineoffset+1)
                 else
                   write('|':i-lastpos);
                 lastpos:= i
               end
             else;
             p:= p^.next;
           end
         else;
         if p <> nil then
           if p^.pos + lineoffset-1 <> lastpos then
             i:= i+1
           else
         else;
       end;
     end;
{******************************************************************}
   procedure underline(p: garbledptr; q: errormark; lastpos: integer);
   { This routine underlines any text on the source line which was discarded
     during the error recovery process. The call parameter "garbledptr" is
     a pointer to a list which contains the start and stop line positions
     for all "junk" that was previously recorded by the "updatesource"
     routine. In this module, it is just a matter of extracting the start
     and stop positions from each node in the list and printing a "%" symbol
     when the incrementing line count is contained within the "junkstart"
     to "junkstop" range. If a junk symbol position coincides with a vertical
     line position (which extends downward from the error posit on the line) then
     the junk symbol is printed to permit clear visual recognition of the
     discarded text. The underlining information is output from a line buffer
     which contains either a blank space, a "%" symbol, or a "|" for each
     line position, beginning with 1 (left edge) through 90 (80 column display
     plus 10 (line offset) for the line numbers. }

   type
        linebuf = packed array[1..displayedge] of char;
   var
```

```
      i : integer;
   line : linebuf;
begin
  if p <> nil then
    begin
      for i:= 1 to displayedge do
        line[i]:= ' ';
      i:= 1;
      repeat
        if p^.junkstart <= p^.junkstop then
          if (i >= p^.junkstart) and (i < p^.junkstop) then
            begin
              line[i]:= '%';
              i:= i+1
            end
          else if i = p^.junkstop then
                  begin
                    line[i]:= '%';
                    i:= i+1;
                    p:= p^.next
                  end
          else i:= i+1
        else if p^.junkstop = 0 then
               if (i >= p^.junkstart) and (i <= lastpos+ lineoffset) then
                  begin
                    line[i]:= '%';
                    i:= i+1
                  end
               else i:= i+1
        else p:= p^.next
      until (p = nil) or (i = displayedge+1);
      i:= 1;
      if q <> nil then
        repeat
          if i = (q^.pos-1+lineoffset) then
            begin
              if line[i] <> '%' then
                line[i]:= '|'
              else;
              if q^.next <> nil then
                if q^.pos <> q^.next^.pos then
                  i:= i+1
                else
              else;
              q:= q^.next
            end
          else
            i:= i+1;
        until (q = nil) or (i = displayedge+1)
      else;
      i:= 1;
      while (i <= displayedge) and
            ((line[i] = ' ') or (line[i] = '|')) do
        i:= i+1;
      if i <> displayedge+1 then
        begin
          for i:= 1 to displayedge-1 do
            write(line[i]);
          writeln(line[displayedge]);
        end
      else
    end
  else
end;
{********************************************************************}
  procedure formatline(p: errormark);
  { Formatline is primarily responsible for the horizontal component
    of the error message lines. These begin at the base of each vertical
    error line and extend to the right through column position 90.
    Since multiple errors may occur on one line, this routine resolves
    conflicts between the vertical bar ("|") and the horizontal bar ("_")
    in those situations where the lines cross, with priority being given
    to the vertical bar. Additionally, this routine also prints the line
    message header "****Error". }

  var
      last,i: integer;
  begin
    printmark(p);
    writeln;
    write('****Error ');
    last:= lineoffset;
    write('|':p^.pos + lineoffset-1 - last);
    last:= p^.pos + lineoffset-1;
    p:= p^.next;
    if last = p^.pos + lineoffset-1 then
      p:= p^.next;
    for i:= last+1 to displayedge do
      if p <> nil then
        if i= p^.pos + lineoffset-1 then
          begin
            write('|');
            p:= p^.next
          end
        else
          write('_')
      else
        write('_')
  end;
{********************************************************************}
  function findtextend;
  { This routine is used to determine the position where actual program
    text terminates on a line to prevent underlining of trailing edge
    comments. }

  var
       found: boolean;
      nested: boolean;
      last,i: integer;
  begin
    with lexx do
      begin
        i:= lastpos;
        last:= lastpos;
        nested:= false;
        found:= false;
        if lastpos > 1 then
          repeat
            if linebuf[i] = ' ' then
              repeat
                i:= i-1
              until (linebuf[i] <> ' ') or (i = 0)
            else if (linebuf[i] = '}') or ((linebuf[i] = ')')
                                    and (linebuf[i-1] = '*')) then
                   begin
                     last:= i;
                     repeat
                       i:= i-1
                     until (linebuf[i] = '{') or ((linebuf[i] = '*')
                            and (linebuf[i-1] = '(')) or (i = 0) or
              ((linebuf[i] = '}') or ((linebuf[i] = ')')
                       and (linebuf[i-1] = '*')));
          if i > 0 then
            if (linebuf[i] = '}') or ((linebuf[i] = ')')
                   and (linebuf[i-1] = '*')) then
             nested:= true
            else if linebuf[i] = '*' then
                i:= i-2
            else
             i:= i-1
          else
         end
      else
       found:= true
     until (found or (i = 0) or nested)
    else
   end;
   if nested then
    findtextend:= last
   else
    findtextend:= i
  end;
{***************************************************************************}
{***************************************************************************}
                 ERROR MESSAGE DRIVER
{***************************************************************************}
  procedure errormessage;
  { This routine coordinates the collection of the error information and
   traversal of each linked list to output the error messages.  This module
   is called by the end-of-line procedure ("endline") immediately after
   printing the line (if the error pointer is not nil).  The code here
   consists primarily those procedure calls required to output the lists and
   the underline buffer(if required).  Prior to returning to the endline
   routine, all error pointers are reset for the next line.}

  var
    errmarker: errormark;
     lastmark: integer;
          p: errorptr;
          q: lexerrorptr;
  begin
   with error do
    begin
```

```
     p:= errptr;
     q:= lexerrptr;
     errmarker:= nil;
     collecterrors(q,p,errmarker,lastmark);
     if garbledlist <> nil then
       underline(garbledlist,errmarker,lexx.textend);
     while errmarker <> nil do
       begin
         with p^.listing,q^.listing do
           begin
             formatline(errmarker);
             if errmarker^.typ = 'l' then
               begin
                 writeln(message);
                 q:= q^.next
               end
             else begin
                    writeln('Bad ','"',diagname: length(diagname),'"');
                    outputhistory(starthist,p,errmarker,lastmark);
                    outputlegal(expected,p,errmarker,lastmark,diagrams);
                    p:= p^.next;
                  end;
           end;
         errmarker:= errmarker^.next;
       end;
     garbledlist:= nil;
     lexerrptr:= nil;
     errptr:= nil;
    end;
  end;
{********************************************************************}
```

```
#include "global.h"
#include "common.h"
```

OUTPUT ROUTINES

```
 {*******************************************************************}
                  ERROR MESSAGE OUTPUT ROUTINES
 {*******************************************************************}
   These routines output the contents of the history and legal lists.  Much
   of the code in the following two modules is very similar, however, Pascal's
   strong typing precludes combining operations involving the different
   types "historyptr" and "legalptr".
{**********************************************************************}
  procedure outputhistory;
  { The history list output consists of writing "Recognized: " followed
    by the name of each syntactic unit which is stored in the history list.
    If the name represents a nonterminal box, then the output will be
    of the form '< name >', as opposed to just 'name' for lexemes.  If the
    list contains more than 6 elements, then only the first 3 and last 3
    will be shown, with three each on either side of the "..." notation.
    As is also the case with the legal list, a line counter is maintained to
    keep track of spacing contraints so that the message remains contained
    within the 132 column boundary.  The constant "justify1" represents the
    field width necessary to position the header, "justify3" for the items in
    the list,  and "space1" and "space2" are used in calculations for the
    right edge boundary.  Finally, since the message may be followed by others
    which pertain to the same line of source text, these routines must access
    the "errormark" list to maintain any required preceding vertical marks which
    are produced by the "printmark" display uitlity. }

  var
     currentpos: integer;
         count: integer;
         total: integer;

    function getlength(p,q: historyptr): integer;

    var  i: integer;

    begin
     i:= 0;
     repeat
       i:= i+1;
```

```
      p:= p^.next
    until (p = q);
    getlength:= i+1
  end; {getlength}

begin
  if r^.next <> nil then
    begin
      printmark(r^.next);
      write('Recognized:  ':justify1-lastmark)
    end
  else
    write('Recognized:  ':justify1);
  currentpos:= justify1+1;
  if p <> nil then
    begin
      with q^.listing do
        if starthist <> endhist then
          begin
            total:= getlength(p,endhist);
            count:= 1;
            repeat
              if (total > maxhistoryitems) and
                 (count = maxhistoryitems-2) then
                begin
                  p^.name:= '...';
                  p^.typ:= lexeme
                end
              else;
              if length(p^.name)+space1 <= lineprintwidth-currentpos then
                begin
                  if p^.typ = lexeme then
                    begin
                      write(p^.name:length(p^.name),' ');
                      currentpos:= currentpos+length(p^.name)+1
                    end
                  else begin
                         write('<',p^.name:length(p^.name),'> ');
                         currentpos:= currentpos+length(p^.name)+3
                       end
                end
              else begin
                     writeln;
                     if r^.next <> nil then
                begin
                  printmark(r^.next);
                  write(' ':justify3-lastmark);
                  if p^.typ = lexeme then
                    begin
                      write(p^.name:length(p^.name),' ');
                      currentpos:= justify3+length(p^.name)+1
                    end
                  else begin
                         write('<',p^.name:length(p^.name),'> ');
                         currentpos:= justify3+length(p^.name)+3
                       end
                end
              else begin
                     write(' ':justify3);
                     if p^.typ = lexeme then
                       begin
                         write(p^.name:length(p^.name),' ');
                         currentpos:= justify3+length(p^.name)+1
                       end
                     else begin
                            write('<',p^.name:length(p^.name),'> ');
                            currentpos:= justify3+length(p^.name)+3
                          end
                   end;
            end;
      if (total > maxhistoryitems) and
         (count = maxhistoryitems-2) then
        while (total-count) >= maxhistoryitems div 2  do
          begin
            count:= count+1;
            p:= p^.next
          end
      else begin
              count:= count+1;
              p:= p^.next
            end
    until (p = endhist)
  end
else;
if length(p^.name)+space1 <= lineprintwidth-currentpos then
  if p^.typ = lexeme then
    writeln(p^.name: length(p^.name))
  else
```

```
               writeln('<',p^.name:length(p^.name),'>')
           else begin
                  writeln;
                  if r^.next <> nil then
                    begin
                      printmark(r^.next);
                      write(' ':justify3-lastmark);
                      if p^.typ = lexeme then
                        writeln(p^.name: length(p^.name))
                      else
                        writeln('<',p^.name:length(p^.name),'>')
                    end
                  else begin
                         write(' ':justify3);
                         if p^.typ = lexeme then
                           writeln(p^.name: length(p^.name))
                         else
                           writeln('<',p^.name:length(p^.name),'>')
                       end
                end
       end
     else
       writeln('nothing yet in ',q^.listing.diagname:
                length(q^.listing.diagname));
   end;
{*********************************************************************}
```

```
procedure outputlegal;
{ This module is much like outputhistory with only a few differences.
  Since the legal list is only concerned with lexemes, the "< >" notation
  is not required, but rather all names are simply shown as "name".  The
  constant "justify2" is computed to properly justify the phrase "Legal
  would have been: ", which is output as a header to the list.  If the
  list requires more than one line, justification reverts to "justify3" in
  order to line up with the history list output.  All items are output
  irregardless of the length of the legal list, since this information may be
  especially important to the novice programmer. }

var
    currentpos: integer;
begin
  if r^.next <> nil then
    begin
      printmark(r^.next);
      write('Legal would have been:  ':justify2-lastmark)
    end
  else
    write('Legal would have been:  ':justify2);
  currentpos:= justify2+1;
  if p^.next <> nil then
    with q^.listing do
      begin
        repeat
          if length(diagrams[p^.boxnum].name)+space1 <=
             lineprintwidth-currentpos then
            begin
              write('"',diagrams[p^.boxnum].name:
                  length(diagrams[p^.boxnum].name),'",');
              currentpos:= currentpos+length(diagrams[p^.boxnum].name)+3
            end
          else begin
                 writeln;
                 if r^.next <> nil then
                   begin
                     printmark(r^.next);
                     write(' ':justify3-lastmark);
                     write('"',diagrams[p^.boxnum].name:
                           length(diagrams[p^.boxnum].name),'",');
                   end
                 else begin
                        write(' ':justify3);
```

```
                           write('"',diagrams[p^.boxnum].name:
                                 length(diagrams[p^.boxnum].name),'",');
                         end;
                    currentpos:= justify3+length(diagrams[p^.boxnum].name)+3;
                  end;
             p:= p^.next
           until (p^.next = nil);
           if length(diagrams[p^.boxnum].name)+space2 <=
              lineprintwidth-currentpos then
             writeln(' or "',diagrams[p^.boxnum].name:
                     length(diagrams[p^.boxnum].name),'"')
           else begin
                  writeln;
                  if r^.next <> nil then
                    begin
                      printmark(r^.next);
                      write(' ':justify3-lastmark);
                      writeln(' or "',diagrams[p^.boxnum].name:
                              length(diagrams[p^.boxnum].name),'"');
                    end
                  else begin
                         write(' ':justify3);
                         writeln(' or "',diagrams[p^.boxnum].name:
                                 length(diagrams[p^.boxnum].name),'"');
                       end
                end
         end
       else
         writeln('"',diagrams[p^.boxnum].name:
                 length(diagrams[p^.boxnum].name),'"')
   end;
{*********************************************************************}
```

```
{***************************************************************************}
                              PRINT UTILITIES
{***************************************************************************}
  { These routines output various messages and debugging information as
    selected by the command line switches.  With the exception of the
    EOF/EOP messages, these features are not operationally part of the
    program, however, they provide convenient aids when experimenting or
    performing maintenance related activities. }
{***************************************************************************}
  function length;
  { Returns the proper field width for the output }

  var
    i: integer;
  begin
   i:= 1;
   while name[i] <> ' ' do
     i:= i+1;
   length:= i-1;
  end;
{***************************************************************************}
  procedure printhist;
  { This procedure prints the contents of the history list if the
    command line switch "printhistory" is activated.  This routine is
    called from procedure "update" after adding a new element. }

  begin
   writeln('History list:');
   writeln;
   while p <> nil do
     begin
       write('  ',p^.name: length(p^.name));
       p:= p^.next
     end;
   writeln;
  end;
{***************************************************************************}
  procedure print;
  { This routine outputs the contents of the stack, and messages for
    end of file, parsing halts, and parsing resumes.  Selection is
    determined based upon one of the following switch call parameters:
    prstack, preof, preop, prhalt, prresume. }
```

```
var
   pos,line: integer;
begin
  with lexx do
    begin
      if list then
        begin
          pos:= lastpos ;
          line:= linenum+1
        end
      else begin
             pos:= chpos-1;
             line:= linenum+1
           end
    end;
  if switch = prstack then
    begin
      writeln;
      writeln('Stack configuration :');
      while p <> nil do
        begin
          write(p^.name);
          if ord(p^.kind) = 0 then
            write('activation')
          else
            write('recovery');
          if p^.kind <> recovery then
            writeln('        ','lasttrue: ',p^.lasttrue:3)
          else
            writeln(' ');
          p:= p^.next
        end;
      writeln;
    end
  else if switch = prhalt then
         begin
           writeln;
           writeln('token=',token);
           writeln('Entered recovery mode at line ',line:3,' pos ',
                           pos:2,' on token "', syntax.name[token]:
                                   length(syntax.name[token]),'"');
           writeln;
         end
   else if switch = prresume then
         begin
           writeln;
           writeln('Resumed parsing at line ',line:3,' pos ',pos:2,
                                  ' on token "', syntax.name[token]:
                                    length(syntax.name[token]),'"');
           writeln;
         end
   else if switch = preof then
         begin
           writeln;
           writeln('   **** Unexpected EOF -- Compilation terminated');
         end
   else if switch = preop then
         begin
           writeln;
           writeln('   **** Detected end of program -- Expected EOF');
         end
   else
  end; {print}
{*********************************************************************}
  procedure printset;
  { This routine is called by the recovery module if the "printrecovset"
   switch is set on the command line. Output includes the name and
   diagram position for each symbol in the recovery set. }

  begin
   writeln('Recovery set:');
   while p <> nil do
    begin
     with p^ do
      writeln('symbol=',name,'  diagposit=',diagrampos:4,
                    '  parentrec=', parentrec^.name);
     p:= p^.next
    end;
   writeln;
  end; {printset}
```

```
{*****************************************************************}
  procedure printsyntax;
  { This routine outputs the contents of the stored transition diagrams
    in response to the command line switch "printbox".  }

  var
      i: integer;
  begin
    for i:= 1 to syntax.total do
      with diagrams[i] do
        begin
          if ord(typ) = 0 then
            begin
              writeln;
              writeln;
              writeln;
              writeln
            end;
          write('box=',i:2,' type=',ord(typ):1,' name=',name,' code='
                ,lexcode:2,' true=',trueptr:2,' false=',falseptr:2);
          writeln(' next=',nextptr:2);
        end
  end;
{*****************************************************************}
```

```
#include "global.h"
#include "common.h"
```

INITIALIZATIONS

```
{**********************************************************************}
                         INITIALIZATION UTILITIES
{**********************************************************************}

  procedure addheadptr(var head: headptr; name: boxname; boxnum: boxptr);
  { This routine is called each time a header box is encountered in the input
    file in order to keep track of where each diagram starts in memory.  This
    information is later applied to the "nextptr" field (recursive pointer) of
    the nonterminal boxes, and is also used during the recovery to find out
    where to recommence parsing if a new activation record needs to be added
    to the existing stack. }

  var
      p,q: headptr;
  begin
    new(p);
    p^.name:= name;
    p^.boxnum:= boxnum;
    p^.next:= nil;
    if head = nil then
      head:= p
    else begin
           q:= head;
           while q^.next <> nil do
             q:= q^.next;
           q^.next:= p
         end
  end;
{*********************************************************************}
  procedure addrestart(var head: restartptr; code: lexvalue;
                             pos: boxptr);
  { This routine is called when a "fiducial" symbol is encountered in the input
    file.  The resultant list is checked during the recovery process to see if
    a fiducial (restart) symbol is present in the input stream. }

  var
     p,q: restartptr;
  begin
    new(p);
  p^.token:= code;
  p^.boxnum:= pos;
  p^.next:= nil;
  if head = nil then
    head:= p
  else begin
         q:= head;
         while q^.next <> nil do
           q:= q^.next;
         q^.next:= p
       end
end;
{**************************************************************************}

procedure addrecov(var head: recovset; code: integer);
{ This routine is called upon encountering a recovery symbol in the input
  file.  A check is included here to prevent duplicate entries since many
  boxes have the same symbol name.  }

var
     p,q: recovset;
begin
  new(p);
  p^.symb:= code;
  p^.next:= nil;
  if head = nil then
    head:= p
  else begin
         q:= head;
         while (q^.next <> nil) and (q^.symb <> code) do
           q:= q^.next;
         if q^.symb <> code then
           q^.next:= p
         else
       end
end;
{**************************************************************************}
```

```
  procedure getname(list: lexemelist; name: boxname; var lexname: syntaxunit);
  { This routine is called by initdiagrams to obtain the syntactic name
   (enumerated type) for an input character string.  The returned name is
   then used to compute the code for a lexeme box. }

  var
      found: boolean;
          i: integer;
  begin
   i:= 1;
   found:= false;
   while not found do
     if name = list[i].id then
       begin
         found:= true;
         lexname:= list[i].su
       end
     else
       i:= i+1;
  end;
{*****************************************************************}
  procedure removespace(var ch: char);
  { Used by the diagram input routine to remove blanks between the
   box data in the input file. }

  begin
   repeat
     read(ch)
   until (ch <> ' ') or eoln(input)
  end;
{*****************************************************************}
```

```
{******************************************************************}
                  LEXICAL INITIALIZATION ROUTINES
{*******************************************************************}
  procedure initlex(var lexx: lexparams);
  { This routine initializes data for the lexical analyzer, including the
    scanner table entries, reserved word list, lexeme name list, and all
    legal Pascal characters. }

  const
      tabch  = 9;
  var
      i,j: integer;
  begin
    with lexx do
      begin

        {initialize reserved words}

       id[0]:= ’do’;              id[1]:= ’if’;
       id[2]:= ’in’;              id[3]:= ’of’;
       id[4]:= ’or’;              id[5]:= ’to’;
       id[6]:= ’and’;             id[7]:= ’div’;
       id[8]:= ’end’;             id[9]:= ’for’;
       id[10]:= ’mod’;            id[11]:= ’nil’;
       id[12]:= ’not’;            id[13]:= ’set’;
       id[14]:= ’var’;            id[15]:= ’case’;
       id[16]:= ’else’;           id[17]:= ’file’;
       id[18]:= ’goto’;           id[19]:= ’then’;
       id[20]:= ’type’;           id[21]:= ’with’;
       id[22]:= ’array’;          id[23]:= ’begin’;
       id[24]:= ’const’;          id[25]:= ’label’;
       id[26]:= ’until’;          id[27]:= ’while’;
       id[28]:= ’write’;          id[29]:= ’downto’;
       id[30]:= ’packed’;         id[31]:= ’record’;
       id[32]:= ’repeat’;         id[33]:= ’forward’;
       id[34]:= ’program’;        id[35]:= ’writeln’;
       id[36]:= ’function’;       id[37]:= ’procedur’;

       idlen[0]:= 0;
       idlen[1]:= 0;
       idlen[2]:= 6;
       idlen[3]:= 15;
       idlen[4]:= 22;
    idlen[5]:= 29;
    idlen[6]:= 33;
    idlen[7]:= 36;
    idlen[8]:= 38;


{ initialize lexeme char name/enumerated type name conversion }


    list1[1].id:= '; ';            list1[1].su:= semicolon;
    list1[2].id:= ', ';            list1[2].su:= comma;
    list1[3].id:= ': ';            list1[3].su:= colon;
    list1[4].id:= '( ';            list1[4].su:= lftparen;
    list1[5].id:= ') ';            list1[5].su:= rtparen;
    list1[6].id:= '= ';            list1[6].su:= equals;
    list1[7].id:= '[ ';            list1[7].su:= lftbracket;
    list1[8].id:= '] ';            list1[8].su:= rtbracket;
    list1[9].id:= '. ';            list1[9].su:= period;
    list1[10].id:= '^ ';           list1[10].su:= pointer;
    list1[11].id:= '..';           list1[11].su:= range;
    list1[12].id:= ':=';           list1[12].su:= becomes;
    list1[13].id:= 'or';           list1[13].su:= orr;
    list1[14].id:= 'of';           list1[14].su:= off;
    list1[15].id:= 'do';           list1[15].su:= doo;
    list1[16].id:= 'in';           list1[16].su:= inn;
    list1[17].id:= 'if';           list1[17].su:= iff;
    list1[18].id:= 'to';           list1[18].su:= too;
    list1[19].id:= 'and';          list1[19].su:= andd;
    list1[20].id:= 'end';          list1[20].su:= endd;
    list1[21].id:= 'set';          list1[21].su:= sett;
    list1[22].id:= 'var';          list1[22].su:= varr;
    list1[23].id:= 'for';          list1[23].su:= forr;
    list1[24].id:= 'mod';          list1[24].su:= modd;
    list1[25].id:= 'div';          list1[25].su:= divv;
    list1[26].id:= 'nil';          list1[26].su:= nill;
    list1[27].id:= 'not';          list1[27].su:= nott;
    list1[28].id:= 'eof';          list1[28].su:= endoffile;

    list2[1].id:= 'else';          list2[1].su:= elsee;
    list2[2].id:= 'then';          list2[2].su:= thenn;
    list2[3].id:= 'with';          list2[3].su:= withh;
    list2[4].id:= 'case';          list2[4].su:= casee;
    list2[5].id:= 'type';          list2[5].su:= typee;
    list2[6].id:= 'file';          list2[6].su:= filee;
    list2[7].id:= 'goto';                    list2[7].su:= gotoo;
    list2[8].id:= 'array';                   list2[8].su:= arrayy;
    list2[9].id:= 'const';                   list2[9].su:= constt;
    list2[10].id:= 'begin';                  list2[10].su:= beginn;
    list2[11].id:= 'while';                  list2[11].su:= whilee;
    list2[12].id:= 'until';                  list2[12].su:= untill;
    list2[13].id:= 'write';                  list2[13].su:= writee;
    list2[14].id:= 'label';                  list2[14].su:= labell;
    list2[15].id:= 'packed';                 list2[15].su:= packedd;
    list2[16].id:= 'repeat';                 list2[16].su:= repeatt;
    list2[17].id:= 'record';                 list2[17].su:= recordd;
    list2[18].id:= 'downto';                 list2[18].su:= downtoo;
    list2[19].id:= 'program';                list2[19].su:= programm;
    list2[20].id:= 'forward';                list2[20].su:= forwardd;
    list2[21].id:= 'function';               list2[21].su:= ffunction;
    list2[22].id:= 'procedure';              list2[22].su:= pprocedure;
    list2[23].id:= 'writeln';                list2[23].su:= writelnn;
    list2[24].id:= 'identifier';             list2[24].su:= identifier;
    list2[25].id:= 'unsigned_real';          list2[25].su:= realconst;
    list2[26].id:= 'endmarker';              list2[26].su:= endmarker;
    list2[27].id:= 'unsigned_integer';       list2[27].su:= intconst;
    list2[28].id:= 'character_string';       list2[28].su:= stringconst;
    list2[29].id:= 'adding_operator';        list2[29].su:= addop;
    list2[30].id:= 'multiplying_operator';   list2[30].su:= mulop;
    list2[31].id:= 'relational_operator';    list2[31].su:= relop;


{initialize scanner table entries}


for i:= 0 to statetotal do
  for j:= 0 to inpsymtotal do
    begin
      tab[i,j]:= -1;
      if i >= 32 then
        begin
          tab[i,9]:= i;
          tab[i,19]:= i;
          tab[i,21]:= i
        end;
      tab[15,j]:= 15;   tab[16,j]:= 15;
      tab[20,j]:= 21;   tab[21,j]:= 21;
      tab[11,13]:= 19;  tab[14,9]:= 18;
    end;
```

```
tab[0,0]:= 1;      tab[0,1]:= 2;      tab[0,2]:= 2;      tab[0,3]:= 3;
tab[0,4]:= 4;      tab[0,5]:= 6;      tab[0,6]:= 7;      tab[0,7]:= 9;
tab[0,8]:= 10;     tab[0,9]:= 11;     tab[0,10]:= 12;    tab[0,11]:= 13;
tab[0,12]:= 14;    tab[0,13]:= 17;    tab[0,14]:= 18;    tab[0,15]:= 19;
tab[0,16]:= 15;    tab[0,17]:= 29;    tab[0,18]:= 20;    tab[0,20]:= 0;
tab[0,21]:= 23;    tab[0,23]:= 29;
tab[3,4]:= 5;      tab[3,5]:= 5;
tab[4,5]:= 5;
tab[7,5]:= 8;
tab[11,9]:= 12;    tab[11,21]:= 34;
tab[14,2]:= 15;
tab[15,2]:= 16;    tab[15,17]:= 0;
tab[16,2]:= 16;
tab[16,13]:= 0;    tab[16,17]:= 0;
tab[20,18]:= 30;
tab[21,18]:= 22;
tab[22,18]:= 21;
tab[23,9]:= 25;    tab[23,19]:= 24;   tab[23,21]:= 23;
tab[24,0]:= 27;    tab[24,9]:= 32;    tab[24,19]:= 32;   tab[24,21]:= 28;
tab[25,0]:= 33;    tab[25,9]:= 31;    tab[25,13]:= 31;
tab[25,19]:= 33;   tab[25,21]:= 26;
tab[26,9]:= 33;    tab[26,19]:= 24;   tab[26,21]:= 26;
tab[27,9]:= 32;    tab[27,19]:= 32;   tab[27,21]:= 28;
tab[28,21]:= 28;
tab[29,23]:= 29;
tab[30,18]:= 21;


   { initialize lexeme representations for table }


tab[0,24]:= ord(stop);           tab[1,24]:= ord(addop);
tab[2,24]:= ord(mulop);           tab[3,24]:= ord(relop);
tab[4,24]:= ord(relop);          tab[5,24]:= ord(relop);
tab[6,24]:= ord(equals);         tab[7,24]:= ord(colon);
tab[8,24]:= ord(becomes);         tab[9,24]:= ord(comma);
tab[10,24]:= ord(semicolon);      tab[11,24]:= ord(period);
tab[12,24]:= ord(range);         tab[13,24]:= ord(pointer);
tab[14,24]:= ord(lftparen);      tab[15,24]:= ord(badcomment);
tab[16,24]:= ord(badcomment);   tab[17,24]:= ord(rtparen);
tab[18,24]:= ord(lftbracket);    tab[19,24]:= ord(rtbracket);
tab[20,24]:= ord(badstring);     tab[21,24]:= ord(badstring);
tab[22,24]:= ord(stringconst);   tab[23,24]:= ord(intconst);
```

```
  tab[24,24]:= ord(badexpon);      tab[25,24]:= ord(baddecpt);
  tab[26,24]:= ord(realconst);     tab[27,24]:= ord(badsign);
  tab[28,24]:= ord(realconst);     tab[29,24]:= ord(illegal);
  tab[30,24]:= ord(zerostring);    tab[31,24]:= ord(intconst);
  tab[32,24]:= ord(badexpart);     tab[33,24]:= ord(baddecimal);
  tab[34,24]:= ord(nodigits);


              {initialize table index characters}


  chrs[0].ch:= ’ ’;  chrs[0].val:= 20;  chrs[1].ch:= ’;’;     chrs[1].val:= 8;
  chrs[2].ch:= ’,’;  chrs[2].val:= 7;   chrs[3].ch:= ’:’;     chrs[3].val:= 6;
  chrs[4].ch:= ’=’;  chrs[4].val:= 5;   chrs[5].ch:= ’(’;     chrs[5].val:= 12;
  chrs[6].ch:= ’)’;  chrs[6].val:= 13;  chrs[7].ch:= ’’’’;    chrs[7].val:= 18;
  chrs[8].ch:= ’>’;  chrs[8].val:= 4;   chrs[9].ch:= ’<’;     chrs[9].val:= 3;
  chrs[10].ch:= ’*’; chrs[10].val:= 2;  chrs[11].ch:= ’/’;    chrs[11].val:= 1;
  chrs[12].ch:= ’+’; chrs[12].val:= 0;  chrs[13].ch:= ’-’;    chrs[13].val:= 0;
  chrs[14].ch:= ’[’; chrs[14].val:= 14; chrs[15].ch:= ’]’;    chrs[15].val:= 15;
  chrs[16].ch:= ’{’; chrs[16].val:= 16; chrs[17].ch:= ’}’;    chrs[17].val:= 17;
  chrs[18].ch:= ’.’; chrs[18].val:= 9;  chrs[19].ch:= ’^’;    chrs[19].val:= 11;
  chrs[20].ch:= ’e’; chrs[20].val:= 19; chrs[21].ch:= ’E’;    chrs[21].val:= 19;
  chrs[22].ch:= ’@’; chrs[22].val:= 11;
  chrs[23].ch:= chr(31);  chrs[23].val:= 10;
  chrs[24].ch:= chr(tabch);   chrs[24].val:= 20;

  end; {with lexx do}
 end;
{********************************************************************}
```

```
{*************************************************************************}
                    SYNTAX INITIALIZATION ROUTINES
{*************************************************************************}
  procedure initdiagrams(var syntax: syntaxdata; var recov: recovdata;
                    var diagrams: syntaxchart; var lexx: lexparams);
  { This is the routine that loads the entire set of syntax diagrams into
    memory from a preconstructed input file.  The algorithm is designed to
    read one diagram box per one line in the input file, and it expects to see
    box data in the following order on the line:  1) relative boxnumber (i.e.
    the header is #1, the first syntactic unit in the diagram is #2, etc.) ,
    2) the box type (header, nonterminal,lexeme,recover,fiducial)  where
    "fiducial" and "recover" are also lexmes, but possess important recovery
    characteristics, 3) the name of the box (as it appears in the drawings),
    4) true exit pointer, and 5) false exit pointer.  Although each diagram is
    a separate entity as far as preparing the input file, the routine saves each
    headptr as it is read, interconnecting the complete set of boxes.  Thus,
    frequent changes may be made, if desired, without necessitating any coding
    changes.  The head pointer of each diagram is then used to compute a "next"
    pointer for all of the nonterminals  (the next pointer for a nonterminal
    tells the parser where to go in order to "expand").
       Warning:  Any line in the file which begins with a number will be regarded
    as a box number, thus beginning a line of data.  Any line not beginning with
    a number is discarded.  }

  const
       numbconvert = 48;
         list1len =  4;
  var
              ch: char;
         chident: char;
        length,i: integer;
       boxnumber: integer;
         lastptr: boxptr;
         numbers: charset;
           names: lexemelist;
       tokenname: syntaxunit;
  begin
    syntax.head:= nil;
    syntax.rstart:= nil;
    recov.symbols:= nil;
    for i:= 0 to totallexemes do
      syntax.name[i]:= '  ';
    boxnumber:= 1;
```

```
numbers:= ['0'..'9'];
repeat
  if not eof(input) then
    if not eoln(input) then
      begin
        removespace(ch);
        if not (ch in numbers) then
          readln(input)
        else with diagrams[boxnumber] do
               begin
                 i:= ord(ch)-numbconvert;
                 read(ch);
                 if ch in numbers then
                   i:= 10*i + ord(ch)-numbconvert;
                 removespace(ch);
                 case ch of
                   'h','H': begin
                              typ:= header;
                              lastptr:= boxnumber;
                            end;
                   'n','N': typ:= nonterminal;
                   'l','L',
                   'f','F',
                   'r','R': typ:= lexeme;
                 end;
                 chident:= ch;
                 repeat
                   read(ch)
                 until (ch = ' ');
                 removespace(ch);
                 for length:= 1 to maxname do
                   name[length]:= ' ';
                 length:= 1;
                 repeat
                   name[length]:= ch;
                   length:= length + 1;
                   read(ch)
                 until (ch = ' ');
                 if typ = header then
                   addheadptr(syntax.head,name,boxnumber)
                 else;
                 if typ = lexeme then
                   begin
                      with lexx do
```

```
                       if length <= list1len
                         then names:= list1
                       else names:= list2;
                     getname(names,name,tokenname);
                     lexcode:= ord(tokenname);
                   end
                 else
                   lexcode:= -1;
                if (chident = 'f') or (chident = 'F') then
                  begin
                    addrestart(syntax.rstart,lexcode,lastptr);
                    addrecov(recov.symbols,lexcode)
                  end;
                if (chident = 'r') or (chident = 'R') then
                  addrecov(recov.symbols,lexcode);
                if name = 'endmarker' then
                  begin
                    diagrams[boxnumber].name:= '. ';
                    syntax.eop:= boxnumber
                  end
                else if name = 'eof' then
                       syntax.last:= boxnumber
                else;
                syntax.name[lexcode]:= diagrams[boxnumber].name;
                case typ of
                  header: nextptr:= boxnumber + 1;
                  lexeme: nextptr:= 0;
                  nonterminal: nextptr:= 0;
                end;
                read(trueptr);
                read(falseptr);
                if (trueptr > 0) and (lastptr > 1) then
                  trueptr:= trueptr + lastptr -1;
                if (falseptr > 0) and (lastptr > 1) then
                  falseptr:= falseptr + lastptr -1;
                boxnumber:= lastptr + i;
              end;
      end
    else
      read(ch)
  else
until eof(input);
syntax.total:= boxnumber-1;
for i:= 1 to syntax.total do
```

```
      with diagrams[i] do
        if typ = nonterminal then
          nextptr:= getheadptr(syntax.head,name)
        else;
    if printbox then
      printsyntax(diagrams,syntax);
  end;
{******************************************************************}
  procedure initvars(var syntax: syntaxdata; var error: errordata;
                     var lexx: lexparams; var diagrams: syntaxchart);
  { This routine initializes various lexical and syntactic variables
    which require a value before commencing syntactic analysis. }

  var
       i: integer;
  begin
    with error do
      begin
        errptr:= nil;
        lexerrptr:= nil;
        garbledlist:=  nil;
      end;
    with syntax do
      begin
        legal:= nil;
        name[ord(illegal)]:= 'illegal_character';
        name[ord(badcomment)]:= 'unclosed_comment'
      end;
    with lexx do
      begin
        letter:= ['A'..'z']-['['..'‘'];
        number:= ['0'..'9'];
        sign:= ['+','-'];
        expon:= ['E','e'];
        limit:= false;
        comments:= false;
        continue:= false;
        badtext:= false;
        count:= 0;
        chpos:= 0;
        linenum:= 0;
        ch:= ' ';
        oldline:= false;
        for i:= 1 to maxline do
          begin
            linebuf[i]:= ' ';
            auxbuf[i]:= ' ';
          end;
        list:= false;
        eol:= eoln(input);
        ch:= getchr(lexx,error,diagrams);
      end;
   end;
{**************************************************************************}
{**************************************************************************}
                    INITIALIZATION DRIVER
{**************************************************************************}
   procedure initialize;
   { This routine directs all lexical and syntactic intitializations. }

   begin
     initlex(lexx);
     reset(input,'syntax.ipt');
     initdiagrams(syntax,recov,diagrams,lexx);
     reset(input,filename);
     initvars(syntax,error,lexx,diagrams);
   end;
{**************************************************************************}
```

TRANSITION DIAGRAM INPUT FILE

This is the input file for the parser which contains the specification for each transition diagram (shown in Appendix B). The information in this file is read by an input routine, storing the information for later use by the parser during syntactic analysis. The following information is contained in the input file:

Box # -- position within the transition diagram, with the header as #1.

Type -- three types of boxes: header, lexeme, nonterminal. If a lexeme is to be designated a either a resynchronization or restart symbol for error recovery, then "recover" is used to specify a resynchronization symbol and "fiducial" is used for the restart symbols.

Name -- name of the box

Trueptr -- true exit path for the box, i.e. which box is next along the true exit path.

Falseptr -- same as for true, but using the false exit path.

Trueptrs or Falseptrs which are associated to either a *return true*, *return false*, or *error exit* are represented in the input file by "-1" for return true, "-2" for return false, and "-3" for an error exit. Comments concerning the input file routine are contained in the initialization section of the listings.

PROGRAM

Box#	Type	Name	Trueptr	Falseptr
1	header	Program	2	2
2	fiducial	program	3	-3
3	lexeme	identifier	4	-3
4	recover	(	5	8
5	lexeme	identifier	6	-3
6	recover	)	8	7
7	recover	,	5	-3
8	recover	;	9	-3
9	nonterminal	block	10	-3
10	recover	endmarker	11	-3
11	recover	eof	-1	-3

BLOCK

Box#	Type	Name	Trueptr	Falseptr
1	header	block	2	2
2	nonterminal	label_declaration	3	8
3	nonterminal	const_declaration	4	4
4	nonterminal	type_declaration	5	5
5	nonterminal	var_declaration	6	6
6	nonterminal	proc/func_declaration	7	7
7	nonterminal	compound_statement	-1	-3
8	nonterminal	const_declaration	4	9
9	nonterminal	type_declaration	5	10
10	nonterminal	var_declaration	6	11
11	nonterminal	proc/func_declaration	7	12
12	nonterminal	compound_statement	-1	-2

LABEL DECLARATION

Box#	Type	Name	Trueptr	Falseptr
1	header	label_declaration	2	2
2	fiducial	label	3	-2
3	lexeme	unsigned_integer	4	-3
4	recover	;	-1	5
5	recover	,	3	-3

CONST DECLARATION

Box#	Type	Name	Trueptr	Falseptr
1	header	const_declaration	2	2
2	fiducial	const	3	-2
3	lexeme	identifier	4	-3
4	recover	=	5	-3
5	nonterminal	constant	6	-3
6	recover	;	7	-3
7	lexeme	identifier	4	-1

TYPE DECLARATION

Box#	Type	Name	Trueptr	Falseptr
1	header	type_declaration	2	2
2	fiducial	type	3	-2
3	lexeme	identifier	4	-3
4	recover	=	5	-3
5	nonterminal	type_denoter	6	-3
6	recover	;	7	-3
7	lexeme	identifier	4	-1

VAR DECLARATION

Box#	Type	Name	Trueptr	Falseptr
1	header	var_declaration	2	2
2	recover	var	3	-2
3	lexeme	identifier	4	-3
4	recover	:	6	5
5	recover	,	3	-3
6	nonterminal	type_denoter	7	-3
7	recover	;	8	-3
8	lexeme	identifier	4	-1

PROCEDURE AND FUNCTION DECLARATION PART

Box#	Type	Name	Trueptr	Falseptr
1	header	proc/func_declaration	2	2
2	fiducial	procedure	3	8
3	lexeme	identifier	4	-3
4	nonterminal	formal_parameter_list	5	5
5	recover	;	6	-3
6	recover	forward	14	7
7	nonterminal	block	14	-3
8	fiducial	function	9	-2
9	lexeme	identifier	10	-3
10	nonterminal	formal_parameter_list	11	12
11	recover	:	13	-3
12	recover	:	13	5
13	lexeme	identifier	5	-3
14	recover	;	15	-3
15	fiducial	procedure	3	16
16	fiducial	function	9	-1

COMPOUND STATEMENT

Box#	Type	Name	Trueptr	Falseptr
1	header	compound_statement	2	2
2	fiducial	begin	3	-2
3	nonterminal	statement	4	4
4	recover	end	-1	5
5	recover	;	3	-3

ORDINAL TYPE

Box#	Type	Name	Trueptr	Falseptr
1	header	ordinal_type	2	2
2	lexeme	identifier	10	3
3	nonterminal	constant	4	6
4	recover	..	5	-3
5	nonterminal	constant	-1	-3
6	recover	(	7	-2
7	lexeme	identifier	8	-3
8	recover	)	-1	9
9	recover	,	7	-3
10	recover	..	5	-1

TYPE DENOTER

Box#	Type	Name	Trueptr	Falseptr
1	header	type_denoter	2	2
2	nonterminal	ordinal_type	-1	3
3	recover	^	4	5
4	lexeme	identifier	-1	-3
5	recover	packed	6	22
6	recover	array	7	13
7	recover	[	8	-3
8	nonterminal	ordinal_type	9	-3
9	recover	]	10	12
10	recover	of	11	-3
11	nonterminal	type_denoter	-1	-3
12	recover	,	8	-3
13	recover	record	14	16
14	nonterminal	field_list	15	-3
15	recover	end	-1	-3
16	recover	set	17	19
17	recover	of	18	-3
18	nonterminal	ordinal_type	-1	-3
19	recover	file	20	-3
20	recover	of	21	-3
21	nonterminal	type_denoter	-1	-3
22	recover	array	7	23
23	recover	record	14	24
24	recover	set	17	25
25	recover	file	20	-2

FIELD LIST

Box#	Type	Name	Trueptr	Falseptr
1	header	field_list	2	2
2	lexeme	identifier	3	9
3	recover	:	5	4
4	recover	,	11	-3
5	nonterminal	type_denoter	6	-3
6	recover	;	10	-1
7	nonterminal	variant_part	8	8
8	recover	;	-1	-1
9	nonterminal	variant_part	8	-1
10	lexeme	identifier	3	7
11	lexeme	identifier	3	-3

VARIANT PART

Box#	Type	Name	Trueptr	Falseptr
1	header	variant_part	2	2
2	recover	case	3	-2
3	lexeme	identifier	4	-3
4	recover	:	5	6
5	lexeme	identifier	6	-3
6	recover	of	7	-3
7	nonterminal	constant	8	-3
8	recover	:	10	9
9	recover	,	7	-3
10	recover	(	11	-3
11	nonterminal	field_list	12	-3
12	recover	)	13	-3
13	recover	;	7	-1

FORMAL PARAMETER LIST

Box#	Type	Name	Trueptr	Falseptr
1	header	formal_parameter_list	2	2
2	recover	(	3	-2
3	recover	var	4	10
4	lexeme	identifier	5	-3
5	recover	:	7	6
6	recover	,	4	-3
7	lexeme	identifier	9	8
8	nonterminal	conformant_array_schema	9	-3
9	recover	)	-1	19
10	lexeme	identifier	5	11
11	recover	procedure	12	14
12	lexeme	identifier	13	-3
13	nonterminal	formal_parameter_list	9	9
14	recover	function	15	-3
15	lexeme	identifier	16	-3
16	nonterminal	formal_parameter_list	17	71
17	recover	:	18	-3
18	lexeme	identifier	9	-3
19	recover	;	3	-3

ACTUAL PARAMETER LIST

Box#	Type	Name	Trueptr	Falseptr
1	header	actual_parameter_list	2	2
2	recover	(	3	-2
3	nonterminal	expression	4	-3
4	recover	)	-1	5
5	recover	,	3	-3

WRITE PARAMETER LIST

Box#	Type	Name	Trueptr	Falseptr
1	header	write_parameter_list	2	2
2	recover	(	3	-2
3	nonterminal	expression	4	-3
4	recover	:	5	8
5	nonterminal	expression	6	-3
6	recover	:	7	8
7	nonterminal	expression	8	-3
8	recover	)	-1	9
9	recover	,	3	-3

VARIABLE ACCESS

Box#	Type	Name	Trueptr	Falseptr
1	header	variable_access	2	2
2	recover	[	3	6
3	nonterminal	expression	4	-3
4	recover	]	9	5
5	recover	,	3	-3
6	recover	.	7	8
7	lexeme	identifier	9	-3
8	recover	^	9	-2
9	recover	[	3	10
10	recover	.	7	11
11	recover	^	9	-1

STATEMENT

Box#	Type	Name	Trueptr	Falseptr
1	header	statement	2	2
2	lexeme	unsigned_integer	3	4
3	recover	:	23	-3
4	lexeme	identifier	5	9
5	nonterminal	actual_parameter_list	-1	6
6	nonterminal	variable_access	7	22
7	recover	:=	8	-3
8	nonterminal	expression	-1	-3
9	fiducial	goto	10	11
10	lexeme	unsigned_integer	-1	-3
11	fiducial	write	12	13
12	nonterminal	write_parameter_list	-1	-3
13	fiducial	writeln	14	15
14	nonterminal	write_parameter_list	-1	-1
15	nonterminal	compound_statement	-1	16
16	nonterminal	if_statement	-1	17
17	nonterminal	case_statement	-1	18
18	nonterminal	repeat_statement	-1	19
19	nonterminal	while_statement	-1	20
20	nonterminal	for_statement	-1	21
21	nonterminal	with_statement	-1	-2
22	recover	:=	8	-1
23	lexeme	identifier	5	24
24	fiducial	goto	10	25
25	fiducial	write	12	26
26	fiducial	writeln	14	27
27	nonterminal	compound_statement	-1	28

28	nonterminal	if_statement	-1	29
29	nonterminal	case_statement	-1	30
30	nonterminal	repeat_statement	-1	31
31	nonterminal	while_statement	-1	32
32	nonterminal	for_statement	-1	33
33	nonterminal	with_statement	-1	-1

EXPRESSION

Box#	Type	Name	Trueptr	Falseptr
1	header	expression	2	2
2	nonterminal	simple_expression	3	-2
3	recover	=	4	5
4	nonterminal	simple_expression	-1	-3
5	recover	relational_operator	4	6
6	recover	in	4	-1

SIMPLE EXPRESSION

Box#	Type	Name	Trueptr	Falseptr
1	header	simple_expression	2	2
2	lexeme	adding_operator	3	5
3	nonterminal	term	4	-3
4	lexeme	adding_operator	3	6
5	nonterminal	term	4	-2
6	recover	or	3	-1

TERM

Box#	Type	Name	Trueptr	Falseptr
1	header	term	2	2
2	nonterminal	factor	3	-2
3	lexeme	multiplying_operator	7	4
4	recover	div	7	5
5	recover	mod	7	6
6	recover	and	7	-1
7	nonterminal	factor	3	-3

FACTOR

Box#	Type	Name	Trueptr	Falseptr
1	header	factor	2	2
2	lexeme	unsigned_integer	-1	3
3	lexeme	unsigned_real	-1	4
4	lexeme	character_string	-1	5
5	recover	nil	-1	6
6	lexeme	identifier	7	9
7	nonterminal	actual_parameter_list	-1	8
8	nonterminal	variable_access	-1	-1
9	recover	[	10	15
10	nonterminal	expression	11	14
11	recover	..	12	13
12	nonterminal	expression	13	-3
13	recover	]	-1	14
14	recover	,	10	-3
15	recover	(	16	18
16	nonterminal	expression	17	-3

17	recover	)	-1	-3
18	recover	not	19	-2
19	nonterminal	factor	-1	-3

CONSTANT

Box#	Type	Name	Trueptr	Falseptr
1	header	constant	2	2
2	lexeme	adding_operator	3	6
3	lexeme	identifier	-1	4
4	lexeme	unsigned_integer	-1	5
5	lexeme	unsigned_real	-1	-3
6	lexeme	identifier	-1	7
7	lexeme	unsigned_integer	-1	8
8	lexeme	unsigned_real	-1	9
9	lexeme	character_string	-1	-2

CONFORMANT ARRAY SCHEMA

Box#	Type	Name	Trueptr	Falseptr
1	header	conformant_array_schema	2	2
2	recover	packed	3	13
3	recover	array	4	-3
4	recover	[	5	-3
5	lexeme	identifier	6	-3
6	recover	..	7	-3
7	lexeme	identifier	8	-3
8	recover	:	9	-3
9	lexeme	identifier	10	-3
10	recover	]	11	-3
11	recover	of	12	-3
12	lexeme	identifier	-1	-3
13	recover	array	14	-2
14	recover	[	15	-3
15	lexeme	identifier	16	-3
16	recover	..	17	-3
17	lexeme	identifier	18	-3
18	recover	:	19	-3
19	lexeme	identifier	21	-3
20	recover	;	15	-3
21	recover	]	22	20
22	recover	of	23	-3
23	lexeme	identifier	-1	24
24	nonterminal	conformant_array_schema	-1	-3

IF STATEMENT

Box#	Type	Name	Trueptr	Falseptr
1	header	if_statement	2	2
2	fiducial	if	3	-2
3	nonterminal	Boolean_expression	4	-3
4	recover	then	5	-3
5	nonterminal	statement	6	6
6	recover	else	7	-1
7	nonterminal	statement	-1	-1

CASE STATEMENT

Box#	Type	Name	Trueptr	Falseptr
1	header	case_statement	2	2
2	recover	case	3	-2
3	nonterminal	expression	4	-3
4	recover	of	5	-3
5	nonterminal	constant	6	-3
6	recover	:	8	7
7	recover	,	5	-3
8	nonterminal	statement	9	9
9	recover	;	11	10
10	recover	end	-1	-3
11	nonterminal	constant	6	10

REPEAT STATEMENT

Box#	Type	Name	Trueptr	Falseptr
1	header	repeat_statement	2	2
2	fiducial	repeat	3	-2
3	nonterminal	statement	4	4
4	recover	until	5	6
5	nonterminal	Boolean_expression	-1	-3
6	recover	;	3	-3

WHILE STATEMENT

Box#	Type	Name	Trueptr	Falseptr
1	header	while_statement	2	2
2	fiducial	while	3	-2
3	nonterminal	Boolean_expression	4	-3
4	recover	do	5	-3
5	nonterminal	statement	-1	-1

FOR STATEMENT

Box#	Type	Name	Trueptr	Falseptr
1	header	for_statement	2	2
2	fiducial	for	3	-2
3	lexeme	identifier	4	-3
4	recover	:=	5	-3
5	nonterminal	expression	6	-3
6	recover	to	7	10
7	nonterminal	expression	8	-3
8	recover	do	9	-3
9	nonterminal	statement	-1	-1
10	recover	downto	7	-3

WITH STATEMENT

Box#	Type	Name	Trueptr	Falseptr
1	header	with_statement	2	2
2	fiducial	with	3	-2
3	lexeme	identifier	4	-3
4	nonterminal	variable_access	5	5
5	recover	do	6	7
6	nonterminal	statement	-1	-1
7	recover	,	3	-3

LIST OF REFERENCES

1. Aho, A.V., and Peterson, T.G., "A Minimum Distance Correction Parser For Context-Free Languages", SIAM Journal of Computing, v. 1, pp. 305-312, 1972.

2. Lyon, G.L., "Syntax-Directed Least-Errors Analysis for Context-Free Languages: A Practical Approach", Communication of the ACM, v. 17, n. 1, pp. 3-13, 1974.

3. Levy, J.P., "Automatic Correction of Syntax Errors in Programming Languages", Acta Informatica, v. 4, pp. 271-292, 1975.

4. Graham S.L. and Rhodes, S.P., "Practical Syntactic Error Recovery", Communications of the ACM, v. 18, n. 11, pp. 639-649, 1975.

5. Pennello, T.J. and DeRemer, F., "A Forward Move Algorithm for LR Error Recovery", Proceedings 5th ACM Symposium on Principles of Programming Languages, pp. 241-254, 1978.

6. Tai, K.C., "Syntactic Error-Correction in Programming Languages", IEEE Transactions on Software Engineering, v. 4, pp. 414-425, 1978.

7. Ripley, G.D. and Druseikis, F.C., "A Statistical Analysis of Syntax Errors", Computer Languages, v. 3, pp. 227-240, 1978.

8. Fischer, C.N., Milton, D.R., and Quiring, S.B., "Efficient LL(1) Error Correction and Recovery Using Only Insertions", Acta Informatica, v. 13, n. 2, pp. 141-154, 1980.

9. Anderson, S.O. and Backhouse, R.C., "An Alternative Implementation of an Insertion-Only Recovery Technique", Acta Informatica, v. 18, pp. 289-298, 1982.

10. Backhouse, R.C., Syntax of Programming Languages: Theory and Practice, London: Prentice-Hall International, 1979.

11. Anderson, S.O., Backhouse, R.C., Bugge, E.H., and Stirling, C.P., "An Assessment of Locally Least-Cost Error Recovery", Computer Journal, v. 26, n. 1, pp. 15-24, 1983.

12. Pai, A.B. and Kieburtz, R.B., "Global Context Recovery: A New Strategy for Syntactic Error Recovery by Table-Driven Parsers", ACM Transactions on Programming Languages and Systems, v. 2, n. 1, 1980.

13. Barnard, D.T. and Holt, R.C., "Hierarchic Syntax Error Repair For LR Grammars", International Journal of Computer and Information Sciences, v. 11, n. 4, pp. 231-257, 1982.

14. Richter, H., "Noncorrecting Syntax Error Recovery", ACM Transactions on Programming Languages and Systems, v. 7, n. 3, pp. 478-489, 1985.

15. Turba, T.N., "An Exception-Based Mechanism for Syntactic Error Recovery", SIGPLAN Notices, v. 19, n. 11, 1984.

16. International Organization for Standardization, Specification for Computer Programming Language Pascal, ISO 7185-1982, 1982.

17. Grogono, P., Programming in Pascal, Addison-Wesley, 1984.

BIBLIOGRAPHY

Aho, A.V. and Ullman, J.D., Principles of Compiler Design, Addison-Wesley, 1979.

Cooper, D., Standard Pascal User Reference Manual, W.W. Norton, Inc., 1983.

Horning, J.J., "What the Compiler Should Tell the User", Compiler Construction: An Advanced Course, 2d ed., pp. 525-548, Springer-Verlag, 1976.

Jensen, K. and Wirth, N., Pascal User Manual and Report, Springer-Verlag, 1985.

Ledgard, H., The American Pascal Standard, Springer-Verlag, 1984.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library (Code 0142) Naval Postgraduate School Monterey, California 93943	2
3.	Chairman (Code 52) Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4.	Computer Technology Programs (Code 37) Naval Postgraduate School Monterey, California 93943	1
5.	Robert W. Floyd Department of Computer Science Margaret Jacks Hall 342 Stanford, California 94305	4
6.	Daniel Davis (Code 52Dv) Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
7.	Lcdr. Paul E. Hallowell 122 Destry Court San Jose, California 95136	5

216625

Thesis
H162122 Hallowell
c.1 Top-down parsing
syntax error recovery.